DesignPatternsPHP Documentation

Release 1.0

Dominik Liebler and contributors

Sep 12, 2018

Contents

This is a collection of known design patterns and some sample code how to implement them in PHP. Every pattern has a small list of examples (most of them from Zend Framework, Symfony2 or Doctrine2 as I'm most familiar with this software).

I think the problem with patterns is that often people do know them but don't know when to apply which.

Patterns

The patterns can be structured in roughly three different categories. Please click on **the title of every pattern's page** for a full explanation of the pattern on Wikipedia.

1.1 Creational
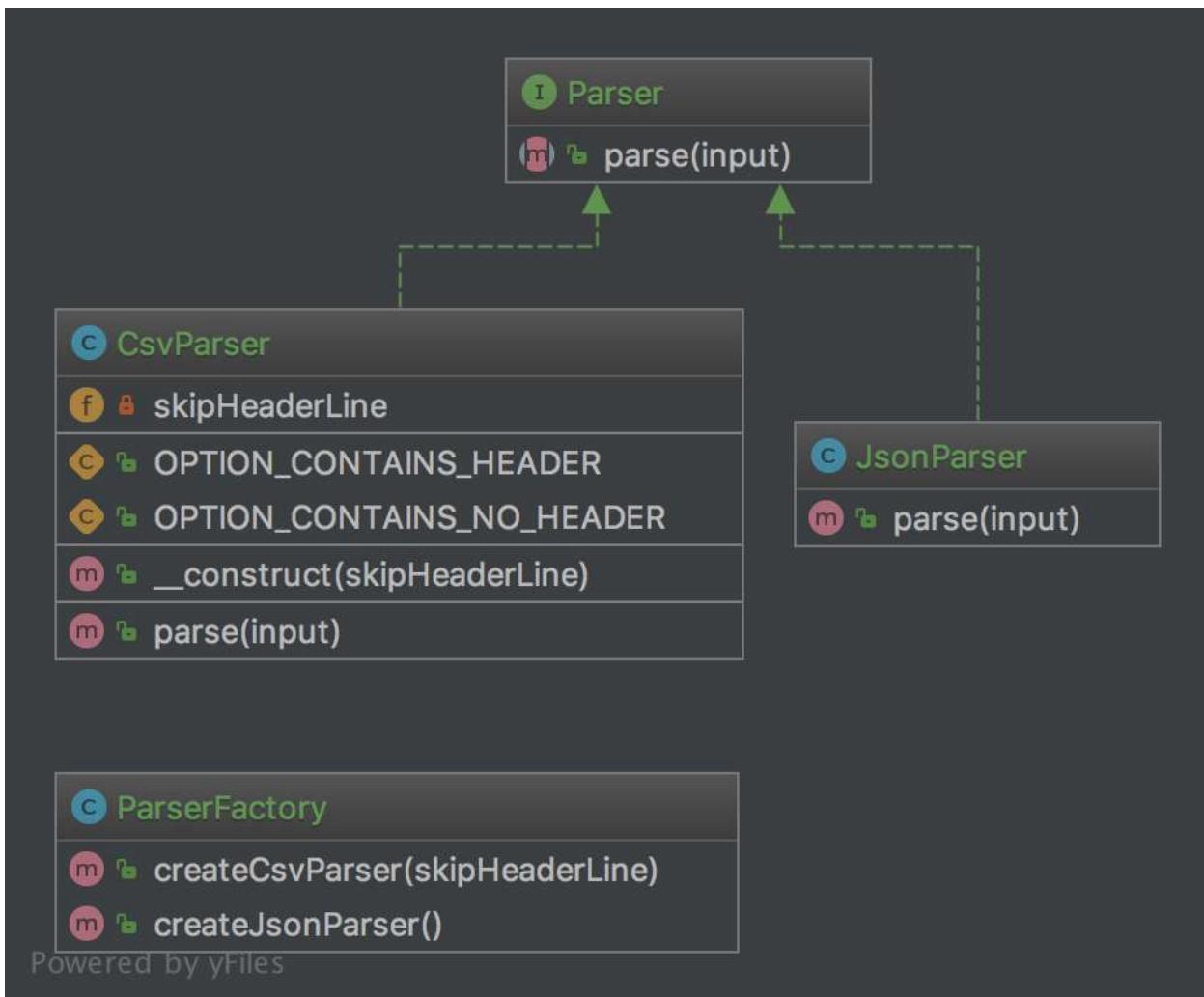
In software engineering, creational design patterns are design patterns that deal with object creation mechanisms, trying to create objects in a manner suitable to the situation. The basic form of object creation could result in design problems or added complexity to the design. Creational design patterns solve this problem by somehow controlling this object creation.

1.1.1 Abstract Factory

Purpose

To create series of related or dependent objects without specifying their concrete classes. Usually the created classes all implement the same interface. The client of the abstract factory does not care about how these objects are created, he just knows how they go together.

UML Diagram

Code

You can also find this code on GitHub

Parser.php

```php
<?php

namespace DesignPatterns\Creational\AbstractFactory;

interface Parser
{
    public function parse(string $input): array;
}
```

CsvParser.php

```php
1   <?php
2
3   namespace DesignPatterns\Creational\AbstractFactory;
4
5   class CsvParser implements Parser
6   {
7       const OPTION_CONTAINS_HEADER = true;
8       const OPTION_CONTAINS_NO_HEADER = false;
9
10      /**
11       * @var bool
12       */
13      private $skipHeaderLine;
14
15      public function __construct(bool $skipHeaderLine)
16      {
17          $this->skipHeaderLine = $skipHeaderLine;
18      }
19
20      public function parse(string $input): array
21      {
22          $headerWasParsed = false;
23          $parsedLines = [];
24
25          foreach (explode(PHP_EOL, $input) as $line) {
26              if (!$headerWasParsed && $this->skipHeaderLine === self::OPTION_CONTAINS_
    ↪HEADER) {
27                  continue;
28              }
29
30              $parsedLines[] = str_getcsv($line);
31          }
32
33          return $parsedLines;
34      }
35  }
```

JsonParser.php

```php
1   <?php
2
3   namespace DesignPatterns\Creational\AbstractFactory;
4
5   class JsonParser implements Parser
6   {
7       public function parse(string $input): array
8       {
9           return json_decode($input, true);
10      }
11  }
```

ParserFactory.php

```php
1   <?php
2
3   namespace DesignPatterns\Creational\AbstractFactory;
4
```

(continues on next page)

```php
5  class ParserFactory
6  {
7      public function createCsvParser(bool $skipHeaderLine): CsvParser
8      {
9          return new CsvParser($skipHeaderLine);
10     }
11
12     public function createJsonParser(): JsonParser
13     {
14         return new JsonParser();
15     }
16 }
```

Test

Tests/AbstractFactoryTest.php

```php
1  <?php
2
3  namespace DesignPatterns\Creational\AbstractFactory\Tests;
4
5  use DesignPatterns\Creational\AbstractFactory\CsvParser;
6  use DesignPatterns\Creational\AbstractFactory\JsonParser;
7  use DesignPatterns\Creational\AbstractFactory\ParserFactory;
8  use PHPUnit\Framework\TestCase;
9
10 class AbstractFactoryTest extends TestCase
11 {
12     public function testCanCreateCsvParser()
13     {
14         $factory = new ParserFactory();
15         $parser = $factory->createCsvParser(CsvParser::OPTION_CONTAINS_HEADER);
16
17         $this->assertInstanceOf(CsvParser::class, $parser);
18     }
19
20     public function testCanCreateJsonParser()
21     {
22         $factory = new ParserFactory();
23         $parser = $factory->createJsonParser();
24
25         $this->assertInstanceOf(JsonParser::class, $parser);
26     }
27 }
```

1.1.2 Builder

Purpose

Builder is an interface that build parts of a complex object.
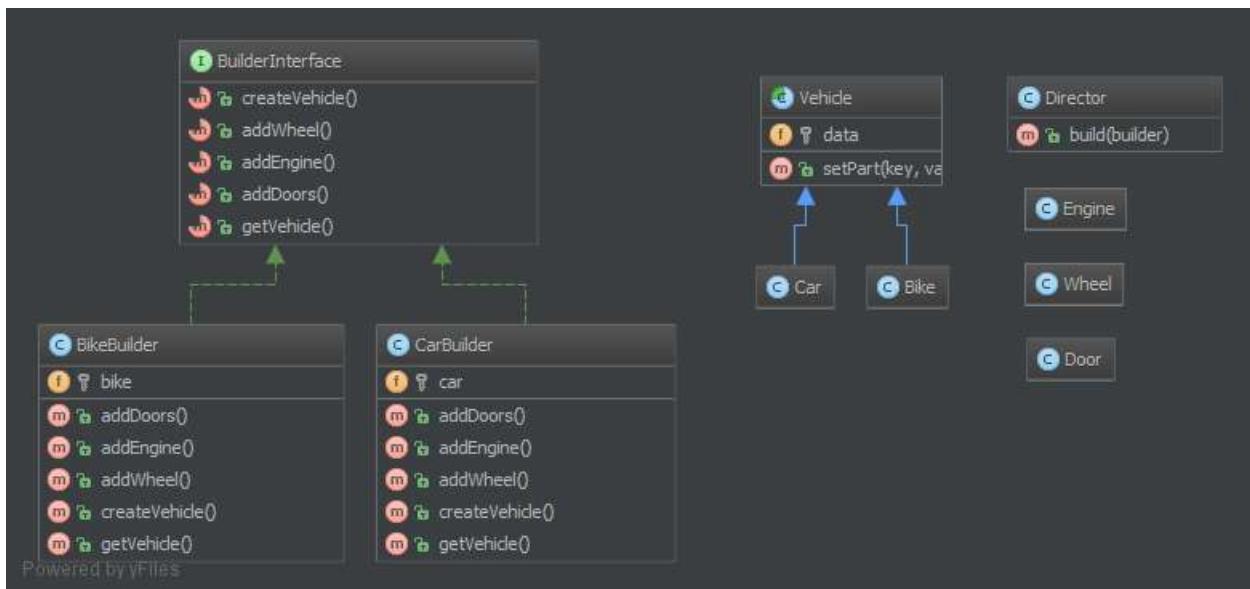
Sometimes, if the builder has a better knowledge of what it builds, this interface could be an abstract class with default methods (aka adapter).

If you have a complex inheritance tree for objects, it is logical to have a complex inheritance tree for builders too.

Note: Builders have often a fluent interface, see the mock builder of PHPUnit for example.

Examples

- PHPUnit: Mock Builder

UML Diagram

Code

You can also find this code on GitHub

Director.php

```php
<?php

namespace DesignPatterns\Creational\Builder;

use DesignPatterns\Creational\Builder\Parts\Vehicle;

/**
 * Director is part of the builder pattern. It knows the interface of the builder
 * and builds a complex object with the help of the builder
 *
 * You can also inject many builders instead of one to build more complex objects
 */
class Director
{
    public function build(BuilderInterface $builder): Vehicle
    {
        $builder->createVehicle();
        $builder->addDoors();
```

(continues on next page)

```php
19          $builder->addEngine();
20          $builder->addWheel();
21
22          return $builder->getVehicle();
23      }
24  }
```

BuilderInterface.php

```php
1   <?php
2
3   namespace DesignPatterns\Creational\Builder;
4
5   use DesignPatterns\Creational\Builder\Parts\Vehicle;
6
7   interface BuilderInterface
8   {
9       public function createVehicle();
10
11      public function addWheel();
12
13      public function addEngine();
14
15      public function addDoors();
16
17      public function getVehicle(): Vehicle;
18  }
```

TruckBuilder.php

```php
1   <?php
2
3   namespace DesignPatterns\Creational\Builder;
4
5   use DesignPatterns\Creational\Builder\Parts\Vehicle;
6
7   class TruckBuilder implements BuilderInterface
8   {
9       /**
10       * @var Parts\Truck
11       */
12      private $truck;
13
14      public function addDoors()
15      {
16          $this->truck->setPart('rightDoor', new Parts\Door());
17          $this->truck->setPart('leftDoor', new Parts\Door());
18      }
19
20      public function addEngine()
21      {
22          $this->truck->setPart('truckEngine', new Parts\Engine());
23      }
24
25      public function addWheel()
26      {
27          $this->truck->setPart('wheel1', new Parts\Wheel());
```

```php
28          $this->truck->setPart('wheel2', new Parts\Wheel());
29          $this->truck->setPart('wheel3', new Parts\Wheel());
30          $this->truck->setPart('wheel4', new Parts\Wheel());
31          $this->truck->setPart('wheel5', new Parts\Wheel());
32          $this->truck->setPart('wheel6', new Parts\Wheel());
33      }
34
35      public function createVehicle()
36      {
37          $this->truck = new Parts\Truck();
38      }
39
40      public function getVehicle(): Vehicle
41      {
42          return $this->truck;
43      }
44  }
```

CarBuilder.php

```php
1   <?php
2
3   namespace DesignPatterns\Creational\Builder;
4
5   use DesignPatterns\Creational\Builder\Parts\Vehicle;
6
7   class CarBuilder implements BuilderInterface
8   {
9       /**
10       * @var Parts\Car
11       */
12      private $car;
13
14      public function addDoors()
15      {
16          $this->car->setPart('rightDoor', new Parts\Door());
17          $this->car->setPart('leftDoor', new Parts\Door());
18          $this->car->setPart('trunkLid', new Parts\Door());
19      }
20
21      public function addEngine()
22      {
23          $this->car->setPart('engine', new Parts\Engine());
24      }
25
26      public function addWheel()
27      {
28          $this->car->setPart('wheelLF', new Parts\Wheel());
29          $this->car->setPart('wheelRF', new Parts\Wheel());
30          $this->car->setPart('wheelLR', new Parts\Wheel());
31          $this->car->setPart('wheelRR', new Parts\Wheel());
32      }
33
34      public function createVehicle()
35      {
36          $this->car = new Parts\Car();
37      }
```

```php
38
39     public function getVehicle(): Vehicle
40     {
41         return $this->car;
42     }
43 }
```

Parts/Vehicle.php

```php
1  <?php
2
3  namespace DesignPatterns\Creational\Builder\Parts;
4
5  abstract class Vehicle
6  {
7      /**
8       * @var object[]
9       */
10     private $data = [];
11
12     /**
13      * @param string $key
14      * @param object $value
15      */
16     public function setPart($key, $value)
17     {
18         $this->data[$key] = $value;
19     }
20 }
```

Parts/Truck.php

```php
1  <?php
2
3  namespace DesignPatterns\Creational\Builder\Parts;
4
5  class Truck extends Vehicle
6  {
7  }
```

Parts/Car.php

```php
1  <?php
2
3  namespace DesignPatterns\Creational\Builder\Parts;
4
5  class Car extends Vehicle
6  {
7  }
```

Parts/Engine.php

```php
1  <?php
2
3  namespace DesignPatterns\Creational\Builder\Parts;
4
```

```php
5  class Engine
6  {
7  }
```

Parts/Wheel.php

```php
1  <?php
2
3  namespace DesignPatterns\Creational\Builder\Parts;
4
5  class Wheel
6  {
7  }
```

Parts/Door.php

```php
1  <?php
2
3  namespace DesignPatterns\Creational\Builder\Parts;
4
5  class Door
6  {
7  }
```

Test

Tests/DirectorTest.php

```php
1  <?php
2
3  namespace DesignPatterns\Creational\Builder\Tests;
4
5  use DesignPatterns\Creational\Builder\Parts\Car;
6  use DesignPatterns\Creational\Builder\Parts\Truck;
7  use DesignPatterns\Creational\Builder\TruckBuilder;
8  use DesignPatterns\Creational\Builder\CarBuilder;
9  use DesignPatterns\Creational\Builder\Director;
10 use PHPUnit\Framework\TestCase;
11
12 class DirectorTest extends TestCase
13 {
14     public function testCanBuildTruck()
15     {
16         $truckBuilder = new TruckBuilder();
17         $newVehicle = (new Director())->build($truckBuilder);
18
19         $this->assertInstanceOf(Truck::class, $newVehicle);
20     }
21
22     public function testCanBuildCar()
23     {
24         $carBuilder = new CarBuilder();
25         $newVehicle = (new Director())->build($carBuilder);
26
27         $this->assertInstanceOf(Car::class, $newVehicle);
```

```
28        }
29  }
```

1.1.3 Factory Method

Purpose
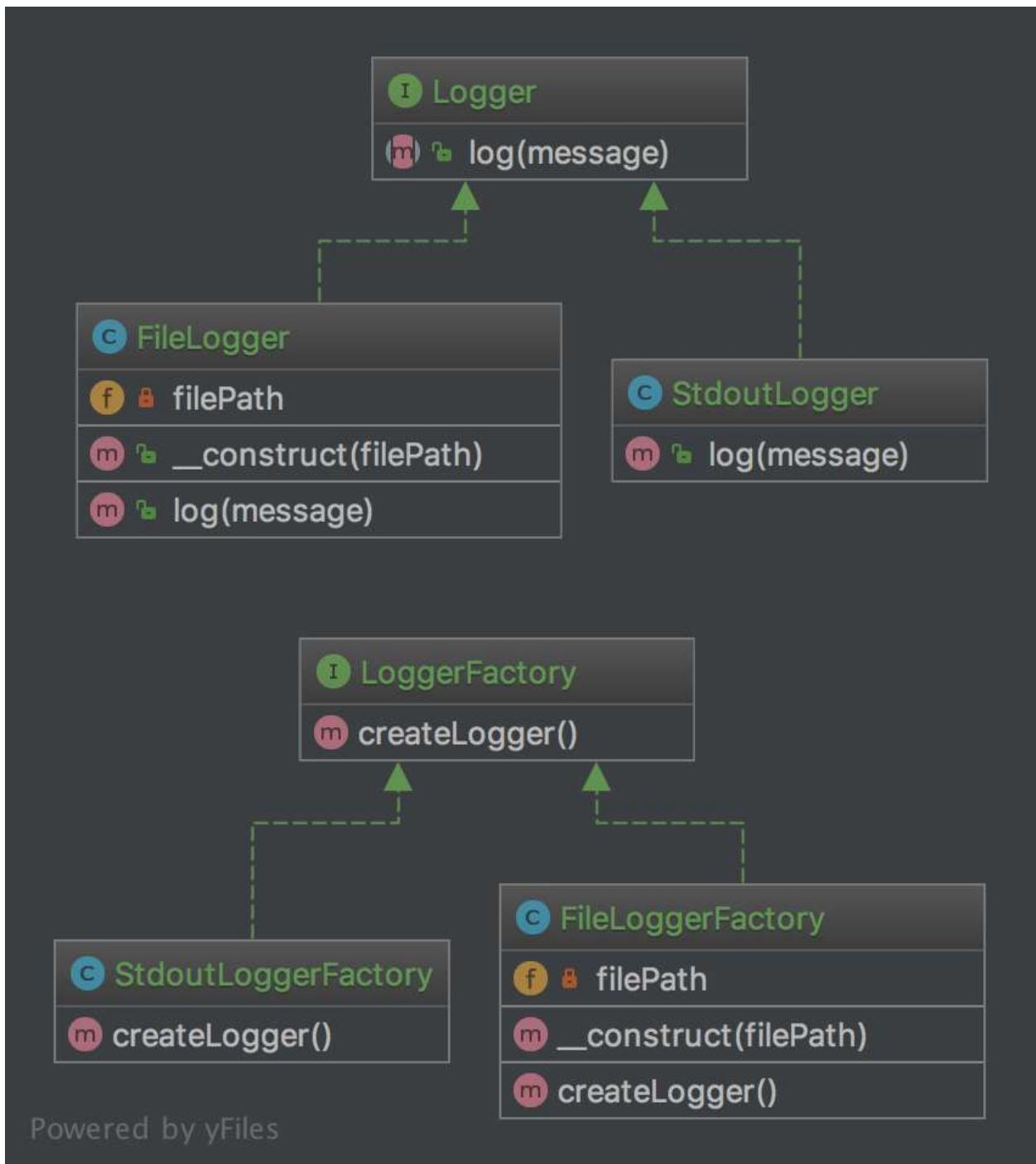
The good point over the SimpleFactory is you can subclass it to implement different ways to create objects.

For simple cases, this abstract class could be just an interface.

This pattern is a "real" Design Pattern because it achieves the Dependency Inversion principle a.k.a the "D" in SOLID principles.

It means the FactoryMethod class depends on abstractions, not concrete classes. This is the real trick compared to SimpleFactory or StaticFactory.

UML Diagram

Code

You can also find this code on GitHub

Logger.php

```php
<?php

namespace DesignPatterns\Creational\FactoryMethod;

interface Logger
{
    public function log(string $message);
}
```

StdoutLogger.php

```php
<?php

namespace DesignPatterns\Creational\FactoryMethod;

class StdoutLogger implements Logger
{
    public function log(string $message)
    {
        echo $message;
    }
}
```

FileLogger.php

```php
<?php

namespace DesignPatterns\Creational\FactoryMethod;

class FileLogger implements Logger
{
    /**
     * @var string
     */
    private $filePath;

    public function __construct(string $filePath)
    {
        $this->filePath = $filePath;
    }

    public function log(string $message)
    {
        file_put_contents($this->filePath, $message . PHP_EOL, FILE_APPEND);
    }
}
```

LoggerFactory.php

```php
<?php

namespace DesignPatterns\Creational\FactoryMethod;

interface LoggerFactory
{
    public function createLogger(): Logger;
}
```

StdoutLoggerFactory.php

```php
<?php

namespace DesignPatterns\Creational\FactoryMethod;

class StdoutLoggerFactory implements LoggerFactory
{
    public function createLogger(): Logger
    {
        return new StdoutLogger();
    }
}
```

FileLoggerFactory.php

```php
<?php

namespace DesignPatterns\Creational\FactoryMethod;

class FileLoggerFactory implements LoggerFactory
{
    /**
     * @var string
     */
    private $filePath;

    public function __construct(string $filePath)
    {
        $this->filePath = $filePath;
    }

    public function createLogger(): Logger
    {
        return new FileLogger($this->filePath);
    }
}
```

Test

Tests/FactoryMethodTest.php

```php
<?php

namespace DesignPatterns\Creational\FactoryMethod\Tests;

use DesignPatterns\Creational\FactoryMethod\FileLogger;
use DesignPatterns\Creational\FactoryMethod\FileLoggerFactory;
use DesignPatterns\Creational\FactoryMethod\StdoutLogger;
use DesignPatterns\Creational\FactoryMethod\StdoutLoggerFactory;
use PHPUnit\Framework\TestCase;

class FactoryMethodTest extends TestCase
{
    public function testCanCreateStdoutLogging()
    {
```

```php
15        $loggerFactory = new StdoutLoggerFactory();
16        $logger = $loggerFactory->createLogger();
17
18        $this->assertInstanceOf(StdoutLogger::class, $logger);
19    }
20
21    public function testCanCreateFileLogging()
22    {
23        $loggerFactory = new FileLoggerFactory(sys_get_temp_dir());
24        $logger = $loggerFactory->createLogger();
25
26        $this->assertInstanceOf(FileLogger::class, $logger);
27    }
28 }
```

1.1.4 Multiton

THIS IS CONSIDERED TO BE AN ANTI-PATTERN! FOR BETTER TESTABILITY AND MAINTAINABILITY USE DEPENDENCY INJECTION!
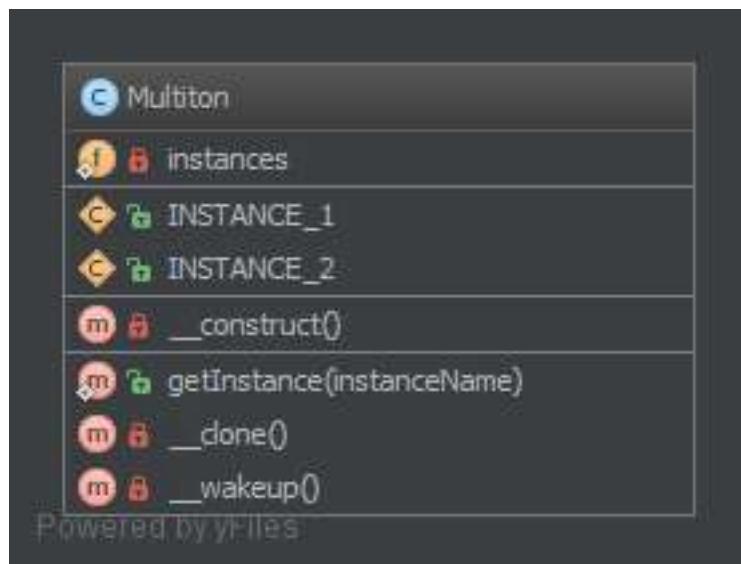
Purpose

To have only a list of named instances that are used, like a singleton but with n instances.

Examples

- 2 DB Connectors, e.g. one for MySQL, the other for SQLite
- multiple Loggers (one for debug messages, one for errors)

UML Diagram

Code

You can also find this code on [GitHub](#)

Multiton.php

```php
<?php

namespace DesignPatterns\Creational\Multiton;

final class Multiton
{
    const INSTANCE_1 = '1';
    const INSTANCE_2 = '2';

    /**
     * @var Multiton[]
     */
    private static $instances = [];

    /**
     * this is private to prevent from creating arbitrary instances
     */
    private function __construct()
    {
    }

    public static function getInstance(string $instanceName): Multiton
    {
        if (!isset(self::$instances[$instanceName])) {
            self::$instances[$instanceName] = new self();
        }

        return self::$instances[$instanceName];
    }

    /**
     * prevent instance from being cloned
     */
    private function __clone()
    {
    }

    /**
     * prevent instance from being unserialized
     */
    private function __wakeup()
    {
    }
}
```

Test

1.1.5 Pool
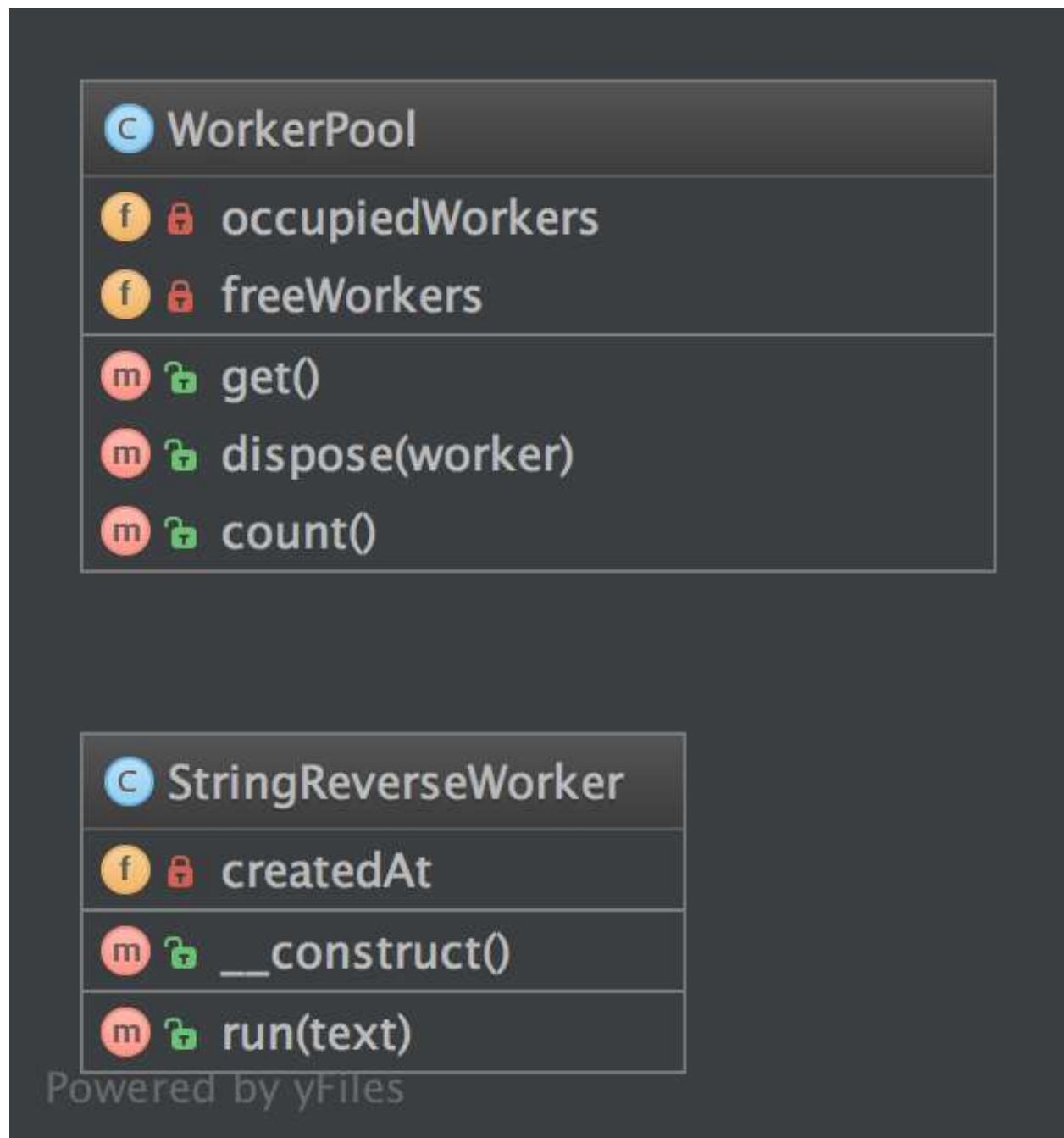
Purpose

The **object pool pattern** is a software creational design pattern that uses a set of initialized objects kept ready to use – a "pool" – rather than allocating and destroying them on demand. A client of the pool will request an object from the pool and perform operations on the returned object. When the client has finished, it returns the object, which is a specific type of factory object, to the pool rather than destroying it.

Object pooling can offer a significant performance boost in situations where the cost of initializing a class instance is high, the rate of instantiation of a class is high, and the number of instances in use at any one time is low. The pooled object is obtained in predictable time when creation of the new objects (especially over network) may take variable time.

However these benefits are mostly true for objects that are expensive with respect to time, such as database connections, socket connections, threads and large graphic objects like fonts or bitmaps. In certain situations, simple object pooling (that hold no external resources, but only occupy memory) may not be efficient and could decrease performance.

UML Diagram

Code

You can also find this code on GitHub

WorkerPool.php

```php
<?php

```

```php
3   namespace DesignPatterns\Creational\Pool;
4
5   class WorkerPool implements \Countable
6   {
7       /**
8        * @var StringReverseWorker[]
9        */
10      private $occupiedWorkers = [];
11
12      /**
13       * @var StringReverseWorker[]
14       */
15      private $freeWorkers = [];
16
17      public function get(): StringReverseWorker
18      {
19          if (count($this->freeWorkers) == 0) {
20              $worker = new StringReverseWorker();
21          } else {
22              $worker = array_pop($this->freeWorkers);
23          }
24
25          $this->occupiedWorkers[spl_object_hash($worker)] = $worker;
26
27          return $worker;
28      }
29
30      public function dispose(StringReverseWorker $worker)
31      {
32          $key = spl_object_hash($worker);
33
34          if (isset($this->occupiedWorkers[$key])) {
35              unset($this->occupiedWorkers[$key]);
36              $this->freeWorkers[$key] = $worker;
37          }
38      }
39
40      public function count(): int
41      {
42          return count($this->occupiedWorkers) + count($this->freeWorkers);
43      }
44  }
```

StringReverseWorker.php

```php
1   <?php
2
3   namespace DesignPatterns\Creational\Pool;
4
5   class StringReverseWorker
6   {
7       /**
8        * @var \DateTime
9        */
10      private $createdAt;
11
12      public function __construct()
```

```php
13      {
14          $this->createdAt = new \DateTime();
15      }
16
17      public function run(string $text)
18      {
19          return strrev($text);
20      }
21  }
```

Test

Tests/PoolTest.php

```php
1   <?php
2
3   namespace DesignPatterns\Creational\Pool\Tests;
4
5   use DesignPatterns\Creational\Pool\WorkerPool;
6   use PHPUnit\Framework\TestCase;
7
8   class PoolTest extends TestCase
9   {
10      public function testCanGetNewInstancesWithGet()
11      {
12          $pool = new WorkerPool();
13          $worker1 = $pool->get();
14          $worker2 = $pool->get();
15
16          $this->assertCount(2, $pool);
17          $this->assertNotSame($worker1, $worker2);
18      }
19
20      public function testCanGetSameInstanceTwiceWhenDisposingItFirst()
21      {
22          $pool = new WorkerPool();
23          $worker1 = $pool->get();
24          $pool->dispose($worker1);
25          $worker2 = $pool->get();
26
27          $this->assertCount(1, $pool);
28          $this->assertSame($worker1, $worker2);
29      }
30  }
```

1.1.6 Prototype
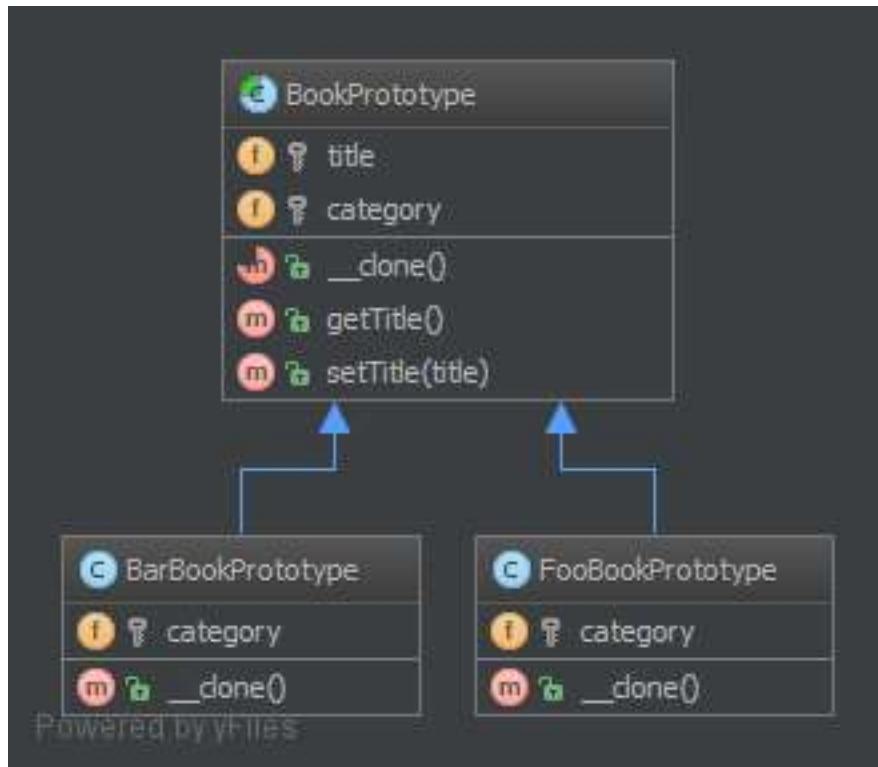
Purpose

To avoid the cost of creating objects the standard way (new Foo()) and instead create a prototype and clone it.

Examples

- Large amounts of data (e.g. create 1,000,000 rows in a database at once via a ORM).

UML Diagram

Code

You can also find this code on GitHub

BookPrototype.php

```php
<?php

namespace DesignPatterns\Creational\Prototype;

abstract class BookPrototype
{
    /**
     * @var string
     */
    protected $title;

    /**
     * @var string
     */
    protected $category;

```

```php
17      abstract public function __clone();
18
19      public function getTitle(): string
20      {
21          return $this->title;
22      }
23
24      public function setTitle($title)
25      {
26          $this->title = $title;
27      }
28  }
```

BarBookPrototype.php

```php
1   <?php
2
3   namespace DesignPatterns\Creational\Prototype;
4
5   class BarBookPrototype extends BookPrototype
6   {
7       /**
8        * @var string
9        */
10      protected $category = 'Bar';
11
12      public function __clone()
13      {
14      }
15  }
```

FooBookPrototype.php

```php
1   <?php
2
3   namespace DesignPatterns\Creational\Prototype;
4
5   class FooBookPrototype extends BookPrototype
6   {
7       /**
8        * @var string
9        */
10      protected $category = 'Foo';
11
12      public function __clone()
13      {
14      }
15  }
```

Test

Tests/PrototypeTest.php

```php
1   <?php
2
```

```
3   namespace DesignPatterns\Creational\Prototype\Tests;
4
5   use DesignPatterns\Creational\Prototype\BarBookPrototype;
6   use DesignPatterns\Creational\Prototype\FooBookPrototype;
7   use PHPUnit\Framework\TestCase;
8
9   class PrototypeTest extends TestCase
10  {
11      public function testCanGetFooBook()
12      {
13          $fooPrototype = new FooBookPrototype();
14          $barPrototype = new BarBookPrototype();
15
16          for ($i = 0; $i < 10; $i++) {
17              $book = clone $fooPrototype;
18              $book->setTitle('Foo Book No ' . $i);
19              $this->assertInstanceOf(FooBookPrototype::class, $book);
20          }
21
22          for ($i = 0; $i < 5; $i++) {
23              $book = clone $barPrototype;
24              $book->setTitle('Bar Book No ' . $i);
25              $this->assertInstanceOf(BarBookPrototype::class, $book);
26          }
27      }
28  }
```

1.1.7 Simple Factory

Purpose

SimpleFactory is a simple factory pattern.

It differs from the static factory because it is not static. Therefore, you can have multiple factories, differently parameterized, you can subclass it and you can mock it. It always should be preferred over a static factory!

UML Diagram

Code

You can also find this code on GitHub

SimpleFactory.php

```php
<?php

namespace DesignPatterns\Creational\SimpleFactory;

class SimpleFactory
{
    public function createBicycle(): Bicycle
    {
        return new Bicycle();
    }
}
```

Bicycle.php

```php
1   <?php
2
3   namespace DesignPatterns\Creational\SimpleFactory;
4
5   class Bicycle
6   {
7       public function driveTo(string $destination)
8       {
9       }
10  }
```

Usage

```php
1   $factory = new SimpleFactory();
2   $bicycle = $factory->createBicycle();
3   $bicycle->driveTo('Paris');
```

Test

Tests/SimpleFactoryTest.php

```php
1   <?php
2
3   namespace DesignPatterns\Creational\SimpleFactory\Tests;
4
5   use DesignPatterns\Creational\SimpleFactory\Bicycle;
6   use DesignPatterns\Creational\SimpleFactory\SimpleFactory;
7   use PHPUnit\Framework\TestCase;
8
9   class SimpleFactoryTest extends TestCase
10  {
11      public function testCanCreateBicycle()
12      {
13          $bicycle = (new SimpleFactory())->createBicycle();
14          $this->assertInstanceOf(Bicycle::class, $bicycle);
15      }
16  }
```

1.1.8 Singleton

THIS IS CONSIDERED TO BE AN ANTI-PATTERN! FOR BETTER TESTABILITY AND MAINTAINABILITY USE DEPENDENCY INJECTION!

Purpose

To have only one instance of this object in the application that will handle all calls.

Examples

- DB Connector

- Logger (may also be a Multiton if there are many log files for several purposes)
- Lock file for the application (there is only one in the filesystem ...)

UML Diagram

Code

You can also find this code on GitHub

Singleton.php

```php
<?php

namespace DesignPatterns\Creational\Singleton;

final class Singleton
{
    /**
     * @var Singleton
     */
    private static $instance;

    /**
     * gets the instance via lazy initialization (created on first usage)
     */
    public static function getInstance(): Singleton
    {
        if (null === static::$instance) {
            static::$instance = new static();
        }

        return static::$instance;
    }

    /**
     * is not allowed to call from outside to prevent from creating multiple
     ↪instances,
     * to use the singleton, you have to obtain the instance from
     ↪Singleton::getInstance() instead
```

```php
27       */
28      private function __construct()
29      {
30      }
31
32      /**
33       * prevent the instance from being cloned (which would create a second instance
    ↪of it)
34       */
35      private function __clone()
36      {
37      }
38
39      /**
40       * prevent from being unserialized (which would create a second instance of it)
41       */
42      private function __wakeup()
43      {
44      }
45  }
```

Test

Tests/SingletonTest.php

```php
1   <?php
2
3   namespace DesignPatterns\Creational\Singleton\Tests;
4
5   use DesignPatterns\Creational\Singleton\Singleton;
6   use PHPUnit\Framework\TestCase;
7
8   class SingletonTest extends TestCase
9   {
10      public function testUniqueness()
11      {
12          $firstCall = Singleton::getInstance();
13          $secondCall = Singleton::getInstance();
14
15          $this->assertInstanceOf(Singleton::class, $firstCall);
16          $this->assertSame($firstCall, $secondCall);
17      }
18  }
```

1.1.9 Static Factory
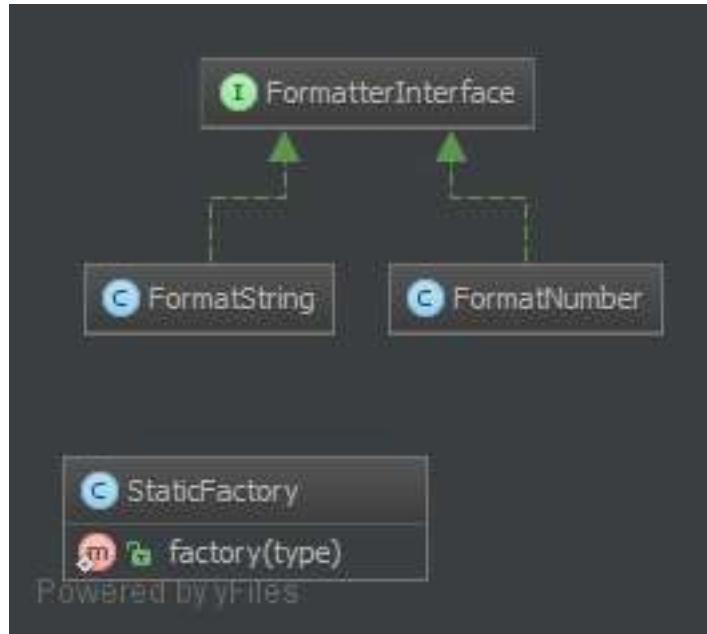
Purpose

Similar to the AbstractFactory, this pattern is used to create series of related or dependent objects. The difference between this and the abstract factory pattern is that the static factory pattern uses just one static method to create all types of objects it can create. It is usually named `factory` or `build`.

Examples

- Zend Framework: `Zend_Cache_Backend` or `_Frontend` use a factory method create cache backends or frontends

UML Diagram

Code

You can also find this code on [GitHub](#)

StaticFactory.php

```php
<?php

namespace DesignPatterns\Creational\StaticFactory;

/**
 * Note1: Remember, static means global state which is evil because it can't be
 →mocked for tests
 * Note2: Cannot be subclassed or mock-upped or have multiple different instances.
 */
final class StaticFactory
{
    /**
     * @param string $type
     *
     * @return FormatterInterface
     */
    public static function factory(string $type): FormatterInterface
    {
        if ($type == 'number') {
            return new FormatNumber();
```

(continues on next page)

```php
20          }
21
22          if ($type == 'string') {
23              return new FormatString();
24          }
25
26          throw new \InvalidArgumentException('Unknown format given');
27      }
28 }
```

FormatterInterface.php

```php
1  <?php
2
3  namespace DesignPatterns\Creational\StaticFactory;
4
5  interface FormatterInterface
6  {
7  }
```

FormatString.php

```php
1  <?php
2
3  namespace DesignPatterns\Creational\StaticFactory;
4
5  class FormatString implements FormatterInterface
6  {
7  }
```

FormatNumber.php

```php
1  <?php
2
3  namespace DesignPatterns\Creational\StaticFactory;
4
5  class FormatNumber implements FormatterInterface
6  {
7  }
```

Test

Tests/StaticFactoryTest.php

```php
1  <?php
2
3  namespace DesignPatterns\Creational\StaticFactory\Tests;
4
5  use DesignPatterns\Creational\StaticFactory\StaticFactory;
6  use PHPUnit\Framework\TestCase;
7
8  class StaticFactoryTest extends TestCase
9  {
10     public function testCanCreateNumberFormatter()
11     {
```

```php
12          $this->assertInstanceOf(
13              'DesignPatterns\Creational\StaticFactory\FormatNumber',
14              StaticFactory::factory('number')
15          );
16      }
17
18      public function testCanCreateStringFormatter()
19      {
20          $this->assertInstanceOf(
21              'DesignPatterns\Creational\StaticFactory\FormatString',
22              StaticFactory::factory('string')
23          );
24      }
25
26      /**
27       * @expectedException \InvalidArgumentException
28       */
29      public function testException()
30      {
31          StaticFactory::factory('object');
32      }
33  }
```

1.2 Structural

In Software Engineering, Structural Design Patterns are Design Patterns that ease the design by identifying a simple way to realize relationships between entities.

1.2.1 Adapter / Wrapper

Purpose
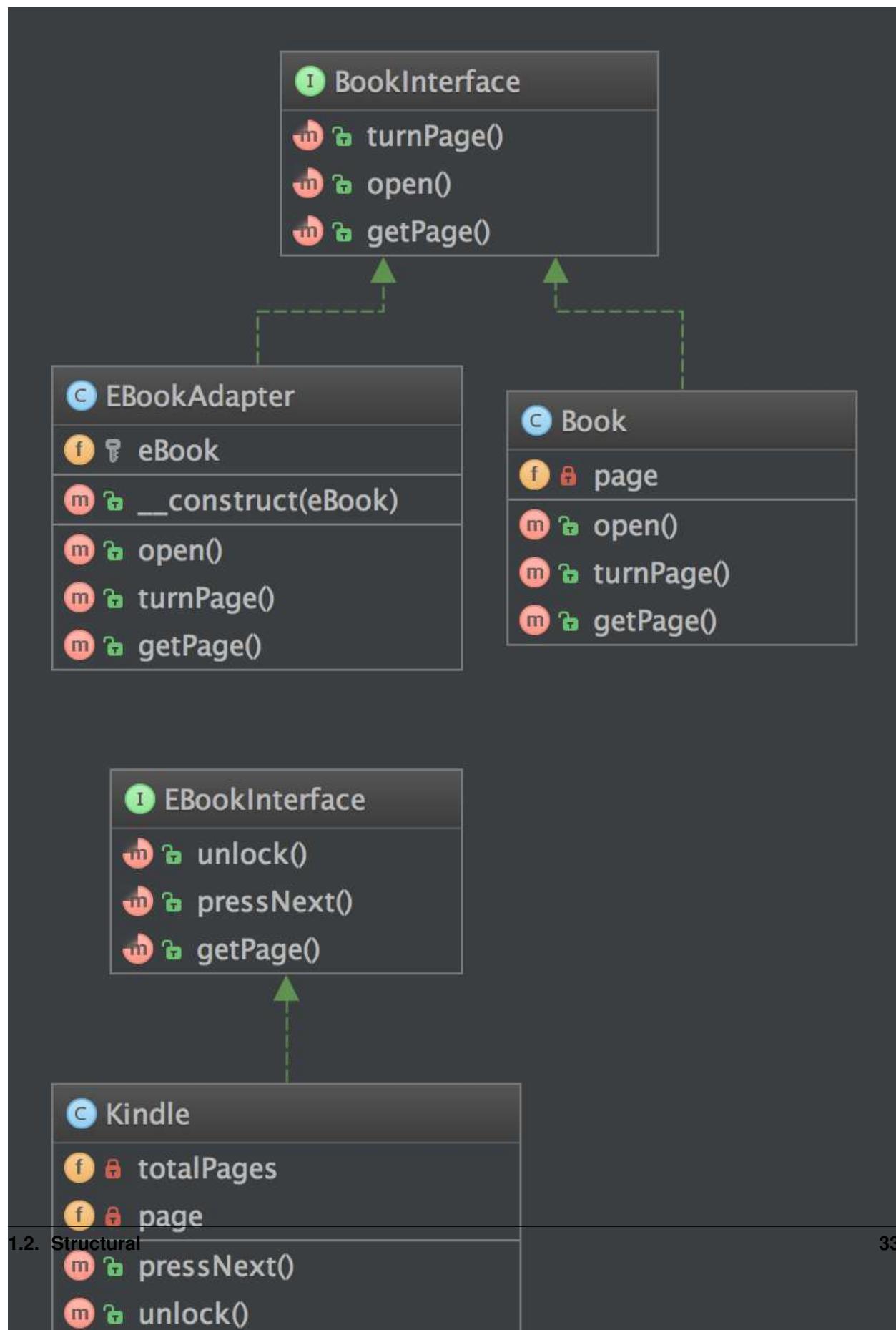
To translate one interface for a class into a compatible interface. An adapter allows classes to work together that normally could not because of incompatible interfaces by providing its interface to clients while using the original interface.

Examples

- DB Client libraries adapter

- using multiple different webservices and adapters normalize data so that the outcome is the same for all

UML Diagram

Code

You can also find this code on GitHub

BookInterface.php

```php
<?php

namespace DesignPatterns\Structural\Adapter;

interface BookInterface
{
    public function turnPage();

    public function open();

    public function getPage(): int;
}
```

Book.php

```php
<?php

namespace DesignPatterns\Structural\Adapter;

class Book implements BookInterface
{
    /**
     * @var int
     */
    private $page;

    public function open()
    {
        $this->page = 1;
    }

    public function turnPage()
    {
        $this->page++;
    }

    public function getPage(): int
    {
        return $this->page;
    }
}
```

EBookAdapter.php

```php
<?php

namespace DesignPatterns\Structural\Adapter;

/**
 * This is the adapter here. Notice it implements BookInterface,
 * therefore you don't have to change the code of the client which is using a Book
 */
```

```php
class EBookAdapter implements BookInterface
{
    /**
     * @var EBookInterface
     */
    protected $eBook;

    /**
     * @param EBookInterface $eBook
     */
    public function __construct(EBookInterface $eBook)
    {
        $this->eBook = $eBook;
    }

    /**
     * This class makes the proper translation from one interface to another.
     */
    public function open()
    {
        $this->eBook->unlock();
    }

    public function turnPage()
    {
        $this->eBook->pressNext();
    }

    /**
     * notice the adapted behavior here: EBookInterface::getPage() will return two
     ↪integers, but BookInterface
     * supports only a current page getter, so we adapt the behavior here
     *
     * @return int
     */
    public function getPage(): int
    {
        return $this->eBook->getPage()[0];
    }
}
```

EBookInterface.php

```php
<?php

namespace DesignPatterns\Structural\Adapter;

interface EBookInterface
{
    public function unlock();

    public function pressNext();

    /**
     * returns current page and total number of pages, like [10, 100] is page 10 of
     ↪100
     *
```

```php
14       * @return int[]
15       */
16      public function getPage(): array;
17  }
```

Kindle.php

```php
1   <?php
2
3   namespace DesignPatterns\Structural\Adapter;
4
5   /**
6    * this is the adapted class. In production code, this could be a class from another
    ↪package, some vendor code.
7    * Notice that it uses another naming scheme and the implementation does something
    ↪similar but in another way
8    */
9   class Kindle implements EBookInterface
10  {
11      /**
12       * @var int
13       */
14      private $page = 1;
15
16      /**
17       * @var int
18       */
19      private $totalPages = 100;
20
21      public function pressNext()
22      {
23          $this->page++;
24      }
25
26      public function unlock()
27      {
28      }
29
30      /**
31       * returns current page and total number of pages, like [10, 100] is page 10 of
    ↪100
32       *
33       * @return int[]
34       */
35      public function getPage(): array
36      {
37          return [$this->page, $this->totalPages];
38      }
39  }
```

Test

Tests/AdapterTest.php

```php
<?php

namespace DesignPatterns\Structural\Adapter\Tests;

use DesignPatterns\Structural\Adapter\Book;
use DesignPatterns\Structural\Adapter\EBookAdapter;
use DesignPatterns\Structural\Adapter\Kindle;
use PHPUnit\Framework\TestCase;

class AdapterTest extends TestCase
{
    public function testCanTurnPageOnBook()
    {
        $book = new Book();
        $book->open();
        $book->turnPage();

        $this->assertEquals(2, $book->getPage());
    }

    public function testCanTurnPageOnKindleLikeInANormalBook()
    {
        $kindle = new Kindle();
        $book = new EBookAdapter($kindle);

        $book->open();
        $book->turnPage();

        $this->assertEquals(2, $book->getPage());
    }
}
```

1.2.2 Bridge
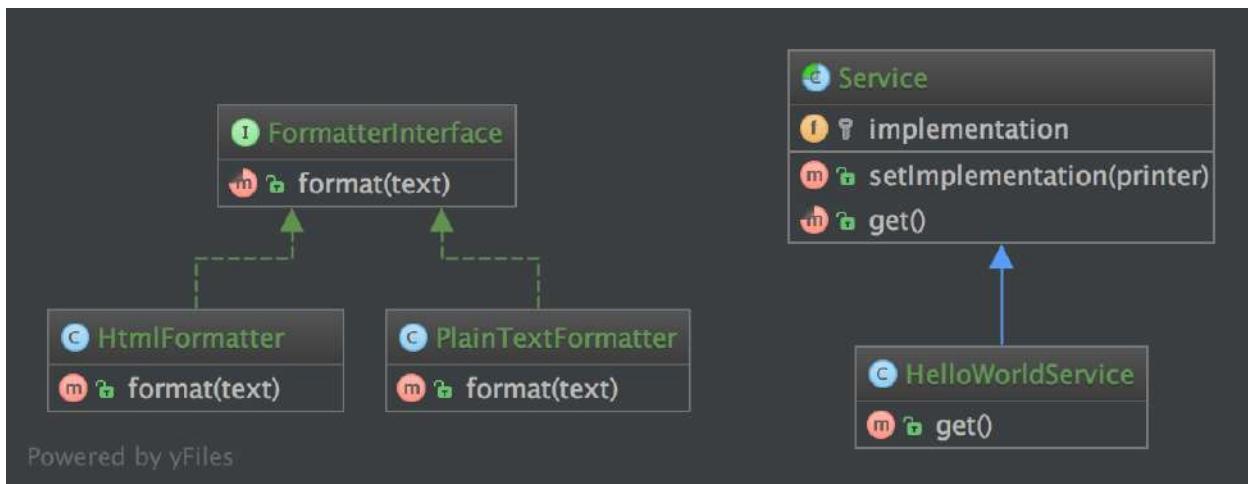
Purpose

Decouple an abstraction from its implementation so that the two can vary independently.

Examples

- Symfony DoctrineBridge

UML Diagram

Code

You can also find this code on GitHub

FormatterInterface.php

```php
<?php

namespace DesignPatterns\Structural\Bridge;

interface FormatterInterface
{
    public function format(string $text);
}
```

PlainTextFormatter.php

```php
<?php

namespace DesignPatterns\Structural\Bridge;

class PlainTextFormatter implements FormatterInterface
{
    public function format(string $text)
    {
        return $text;
    }
}
```

HtmlFormatter.php

```php
<?php

namespace DesignPatterns\Structural\Bridge;

class HtmlFormatter implements FormatterInterface
{
```

```php
7        public function format(string $text)
8        {
9            return sprintf('<p>%s</p>', $text);
10       }
11   }
```

Service.php

```php
1    <?php
2
3    namespace DesignPatterns\Structural\Bridge;
4
5    abstract class Service
6    {
7        /**
8         * @var FormatterInterface
9         */
10       protected $implementation;
11
12       /**
13        * @param FormatterInterface $printer
14        */
15       public function __construct(FormatterInterface $printer)
16       {
17           $this->implementation = $printer;
18       }
19
20       /**
21        * @param FormatterInterface $printer
22        */
23       public function setImplementation(FormatterInterface $printer)
24       {
25           $this->implementation = $printer;
26       }
27
28       abstract public function get();
29   }
```

HelloWorldService.php

```php
1    <?php
2
3    namespace DesignPatterns\Structural\Bridge;
4
5    class HelloWorldService extends Service
6    {
7        public function get()
8        {
9            return $this->implementation->format('Hello World');
10       }
11   }
```

Test

Tests/BridgeTest.php

```php
1  <?php
2
3  namespace DesignPatterns\Structural\Bridge\Tests;
4
5  use DesignPatterns\Structural\Bridge\HelloWorldService;
6  use DesignPatterns\Structural\Bridge\HtmlFormatter;
7  use DesignPatterns\Structural\Bridge\PlainTextFormatter;
8  use PHPUnit\Framework\TestCase;
9
10 class BridgeTest extends TestCase
11 {
12     public function testCanPrintUsingThePlainTextPrinter()
13     {
14         $service = new HelloWorldService(new PlainTextFormatter());
15         $this->assertEquals('Hello World', $service->get());
16
17         // now change the implementation and use the HtmlFormatter instead
18         $service->setImplementation(new HtmlFormatter());
19         $this->assertEquals('<p>Hello World</p>', $service->get());
20     }
21 }
```

1.2.3 Composite
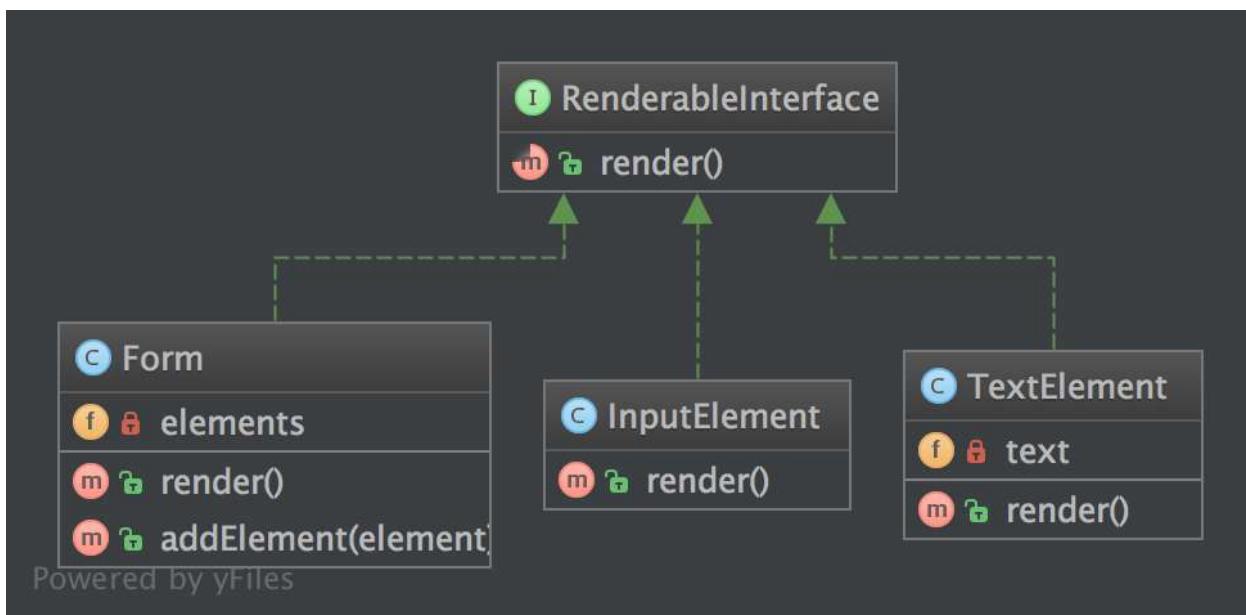
Purpose

To treat a group of objects the same way as a single instance of the object.

Examples

- a form class instance handles all its form elements like a single instance of the form, when `render()` is called, it subsequently runs through all its child elements and calls `render()` on them
- `Zend_Config`: a tree of configuration options, each one is a `Zend_Config` object itself

UML Diagram

Code

You can also find this code on GitHub

RenderableInterface.php

```php
<?php

namespace DesignPatterns\Structural\Composite;

interface RenderableInterface
{
    public function render(): string;
}
```

Form.php

```php
<?php

namespace DesignPatterns\Structural\Composite;

/**
 * The composite node MUST extend the component contract. This is mandatory for
 ↪building
 * a tree of components.
 */
class Form implements RenderableInterface
{
    /**
     * @var RenderableInterface[]
     */
    private $elements;
```

```php
16      /**
17       * runs through all elements and calls render() on them, then returns the␣
   ↪complete representation
18       * of the form.
19       *
20       * from the outside, one will not see this and the form will act like a single␣
   ↪object instance
21       *
22       * @return string
23       */
24      public function render(): string
25      {
26          $formCode = '<form>';
27
28          foreach ($this->elements as $element) {
29              $formCode .= $element->render();
30          }
31
32          $formCode .= '</form>';
33
34          return $formCode;
35      }
36
37      /**
38       * @param RenderableInterface $element
39       */
40      public function addElement(RenderableInterface $element)
41      {
42          $this->elements[] = $element;
43      }
44  }
```

InputElement.php

```php
1   <?php
2
3   namespace DesignPatterns\Structural\Composite;
4
5   class InputElement implements RenderableInterface
6   {
7       public function render(): string
8       {
9           return '<input type="text" />';
10      }
11  }
```

TextElement.php

```php
1   <?php
2
3   namespace DesignPatterns\Structural\Composite;
4
5   class TextElement implements RenderableInterface
6   {
7       /**
8        * @var string
9        */
```

```php
10      private $text;
11
12      public function __construct(string $text)
13      {
14          $this->text = $text;
15      }
16
17      public function render(): string
18      {
19          return $this->text;
20      }
21  }
```

Test

Tests/CompositeTest.php

```php
1   <?php
2
3   namespace DesignPatterns\Structural\Composite\Tests;
4
5   use DesignPatterns\Structural\Composite;
6   use PHPUnit\Framework\TestCase;
7
8   class CompositeTest extends TestCase
9   {
10      public function testRender()
11      {
12          $form = new Composite\Form();
13          $form->addElement(new Composite\TextElement('Email:'));
14          $form->addElement(new Composite\InputElement());
15          $embed = new Composite\Form();
16          $embed->addElement(new Composite\TextElement('Password:'));
17          $embed->addElement(new Composite\InputElement());
18          $form->addElement($embed);
19
20          // This is just an example, in a real world scenario it is important to
    →remember that web browsers do not
21          // currently support nested forms
22
23          $this->assertEquals(
24              '<form>Email:<input type="text" /><form>Password:<input type="text" /></
    →form></form>',
25              $form->render()
26          );
27      }
28  }
```

1.2.4 Data Mapper

Purpose

A Data Mapper, is a Data Access Layer that performs bidirectional transfer of data between a persistent data store (often a relational database) and an in memory data representation (the domain layer). The goal of the pattern is
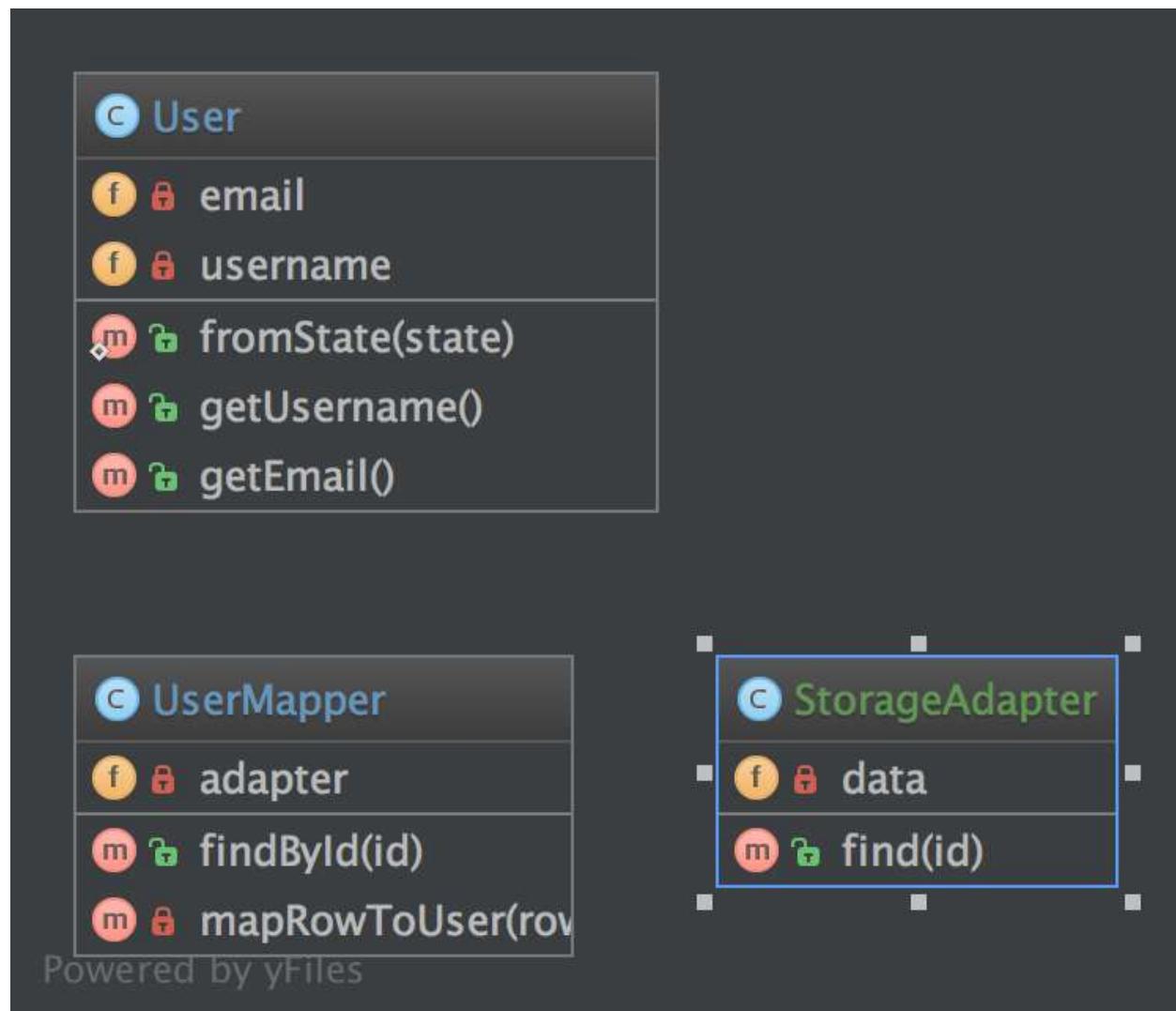
to keep the in memory representation and the persistent data store independent of each other and the data mapper itself. The layer is composed of one or more mappers (or Data Access Objects), performing the data transfer. Mapper implementations vary in scope. Generic mappers will handle many different domain entity types, dedicated mappers will handle one or a few.

The key point of this pattern is, unlike Active Record pattern, the data model follows Single Responsibility Principle.

Examples

- DB Object Relational Mapper (ORM) : Doctrine2 uses DAO named as "EntityRepository"

UML Diagram

Code

You can also find this code on GitHub

User.php

```php
1   <?php
2
3   namespace DesignPatterns\Structural\DataMapper;
4
5   class User
6   {
7       /**
8        * @var string
9        */
10      private $username;
11
12      /**
13       * @var string
14       */
15      private $email;
16
17      public static function fromState(array $state): User
18      {
19          // validate state before accessing keys!
20
21          return new self(
22              $state['username'],
23              $state['email']
24          );
25      }
26
27      public function __construct(string $username, string $email)
28      {
29          // validate parameters before setting them!
30
31          $this->username = $username;
32          $this->email = $email;
33      }
34
35      /**
36       * @return string
37       */
38      public function getUsername()
39      {
40          return $this->username;
41      }
42
43      /**
44       * @return string
45       */
46      public function getEmail()
47      {
48          return $this->email;
49      }
50  }
```

UserMapper.php

```php
1   <?php
2
3   namespace DesignPatterns\Structural\DataMapper;
4
```

```php
class UserMapper
{
    /**
     * @var StorageAdapter
     */
    private $adapter;

    /**
     * @param StorageAdapter $storage
     */
    public function __construct(StorageAdapter $storage)
    {
        $this->adapter = $storage;
    }

    /**
     * finds a user from storage based on ID and returns a User object located
     * in memory. Normally this kind of logic will be implemented using the
     *Repository pattern.
     * However the important part is in mapRowToUser() below, that will create a
     *business object from the
     * data fetched from storage
     *
     * @param int $id
     *
     * @return User
     */
    public function findById(int $id): User
    {
        $result = $this->adapter->find($id);

        if ($result === null) {
            throw new \InvalidArgumentException("User #$id not found");
        }

        return $this->mapRowToUser($result);
    }

    private function mapRowToUser(array $row): User
    {
        return User::fromState($row);
    }
}
```

StorageAdapter.php

```php
<?php

namespace DesignPatterns\Structural\DataMapper;

class StorageAdapter
{
    /**
     * @var array
     */
    private $data = [];
```

```php
12      public function __construct(array $data)
13      {
14          $this->data = $data;
15      }
16
17      /**
18       * @param int $id
19       *
20       * @return array|null
21       */
22      public function find(int $id)
23      {
24          if (isset($this->data[$id])) {
25              return $this->data[$id];
26          }
27
28          return null;
29      }
30  }
```

Test

Tests/DataMapperTest.php

```php
1   <?php
2
3   namespace DesignPatterns\Structural\DataMapper\Tests;
4
5   use DesignPatterns\Structural\DataMapper\StorageAdapter;
6   use DesignPatterns\Structural\DataMapper\User;
7   use DesignPatterns\Structural\DataMapper\UserMapper;
8   use PHPUnit\Framework\TestCase;
9
10  class DataMapperTest extends TestCase
11  {
12      public function testCanMapUserFromStorage()
13      {
14          $storage = new StorageAdapter([1 => ['username' => 'domnikl', 'email' =>
    ↪'liebler.dominik@gmail.com']]);
15          $mapper = new UserMapper($storage);
16
17          $user = $mapper->findById(1);
18
19          $this->assertInstanceOf(User::class, $user);
20      }
21
22      /**
23       * @expectedException \InvalidArgumentException
24       */
25      public function testWillNotMapInvalidData()
26      {
27          $storage = new StorageAdapter([]);
28          $mapper = new UserMapper($storage);
29
30          $mapper->findById(1);
```

```
31        }
32  }
```
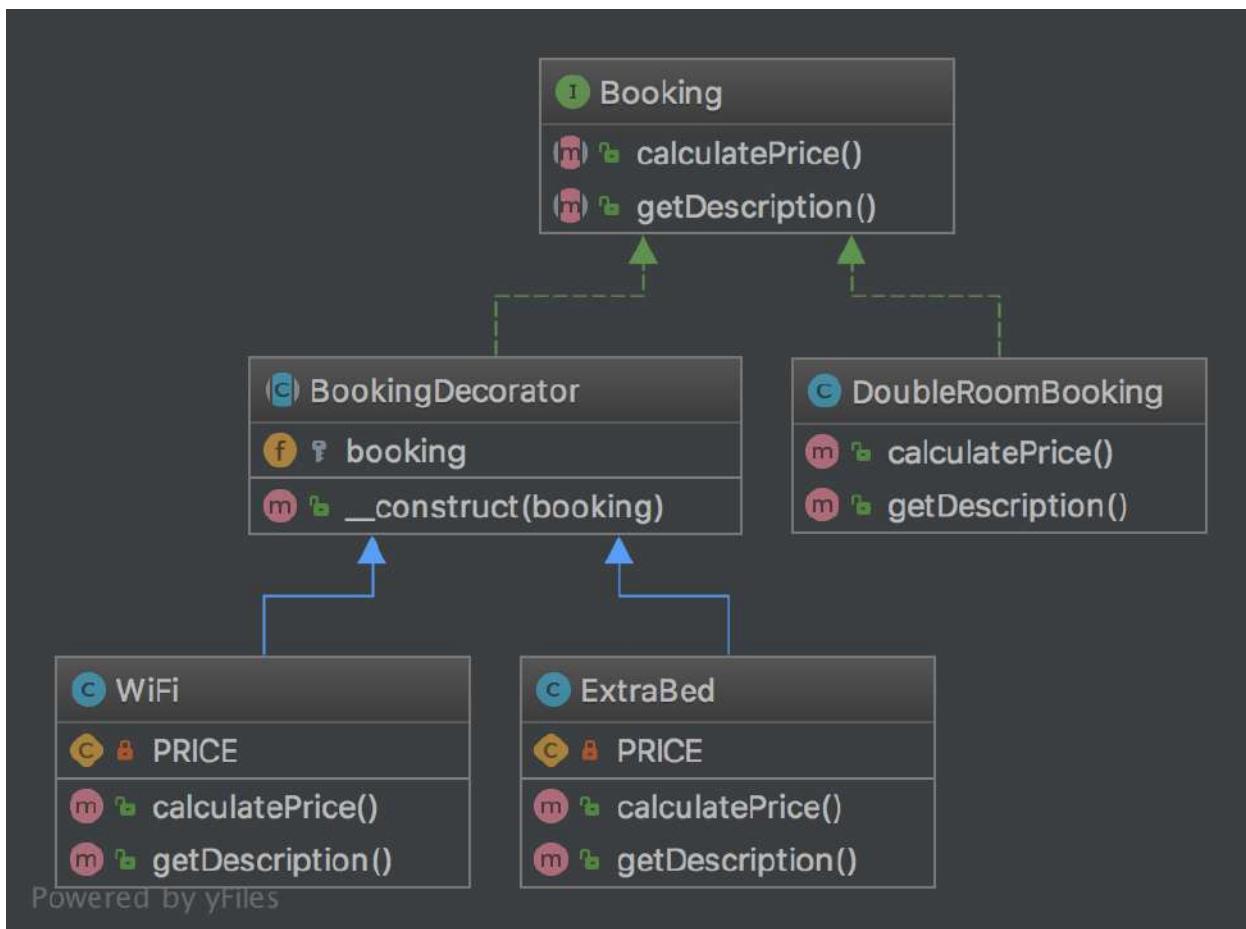
1.2.5 Decorator

Purpose

To dynamically add new functionality to class instances.

Examples

- Zend Framework: decorators for `Zend_Form_Element` instances
- Web Service Layer: Decorators JSON and XML for a REST service (in this case, only one of these should be allowed of course)

UML Diagram

Code

You can also find this code on GitHub

Booking.php

```php
<?php

namespace DesignPatterns\Structural\Decorator;

interface Booking
{
    public function calculatePrice(): int;

    public function getDescription(): string;
}
```

BookingDecorator.php

```php
<?php

namespace DesignPatterns\Structural\Decorator;

abstract class BookingDecorator implements Booking
{
    /**
     * @var Booking
     */
    protected $booking;

    public function __construct(Booking $booking)
    {
        $this->booking = $booking;
    }
}
```

DoubleRoomBooking.php

```php
<?php

namespace DesignPatterns\Structural\Decorator;

class DoubleRoomBooking implements Booking
{
    public function calculatePrice(): int
    {
        return 40;
    }

    public function getDescription(): string
    {
        return 'double room';
    }
}
```

ExtraBed.php

```php
1  <?php
2
3  namespace DesignPatterns\Structural\Decorator;
4
5  class ExtraBed extends BookingDecorator
6  {
7      private const PRICE = 30;
8
9      public function calculatePrice(): int
10     {
11         return $this->booking->calculatePrice() + self::PRICE;
12     }
13
14     public function getDescription(): string
15     {
16         return $this->booking->getDescription() . ' with extra bed';
17     }
18 }
```

WiFi.php

```php
1  <?php
2
3  namespace DesignPatterns\Structural\Decorator;
4
5  class WiFi extends BookingDecorator
6  {
7      private const PRICE = 2;
8
9      public function calculatePrice(): int
10     {
11         return $this->booking->calculatePrice() + self::PRICE;
12     }
13
14     public function getDescription(): string
15     {
16         return $this->booking->getDescription() . ' with wifi';
17     }
18 }
```

Test

Tests/DecoratorTest.php

```php
1  <?php
2
3  namespace DesignPatterns\Structural\Decorator\Tests;
4
5  use DesignPatterns\Structural\Decorator\DoubleRoomBooking;
6  use DesignPatterns\Structural\Decorator\ExtraBed;
7  use DesignPatterns\Structural\Decorator\WiFi;
8  use PHPUnit\Framework\TestCase;
9
10 class DecoratorTest extends TestCase
11 {
12     public function testCanCalculatePriceForBasicDoubleRoomBooking()
```

(continues on next page)

```
13      {
14          $booking = new DoubleRoomBooking();
15
16          $this->assertEquals(40, $booking->calculatePrice());
17          $this->assertEquals('double room', $booking->getDescription());
18      }
19
20      public function testCanCalculatePriceForDoubleRoomBookingWithWiFi()
21      {
22          $booking = new DoubleRoomBooking();
23          $booking = new WiFi($booking);
24
25          $this->assertEquals(42, $booking->calculatePrice());
26          $this->assertEquals('double room with wifi', $booking->getDescription());
27      }
28
29      public function testCanCalculatePriceForDoubleRoomBookingWithWiFiAndExtraBed()
30      {
31          $booking = new DoubleRoomBooking();
32          $booking = new WiFi($booking);
33          $booking = new ExtraBed($booking);
34
35          $this->assertEquals(72, $booking->calculatePrice());
36          $this->assertEquals('double room with wifi with extra bed', $booking->
    →getDescription());
37      }
38  }
```

1.2.6 Dependency Injection

Purpose

To implement a loosely coupled architecture in order to get better testable, maintainable and extendable code.
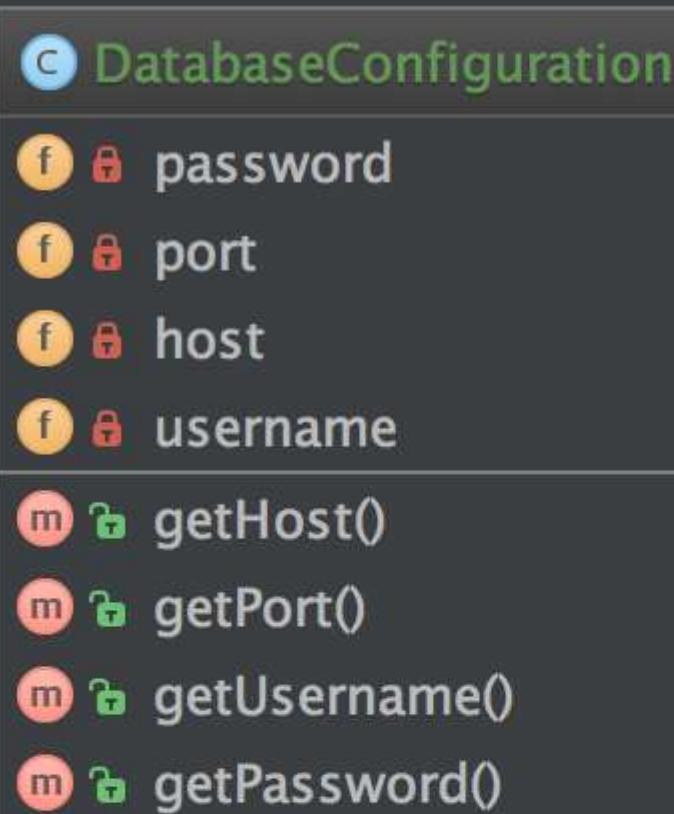
Usage

`DatabaseConfiguration` gets injected and `DatabaseConnection` will get all that it needs from `$config`. Without DI, the configuration would be created directly in `DatabaseConnection`, which is not very good for testing and extending it.

Examples

- The Doctrine2 ORM uses dependency injection e.g. for configuration that is injected into a `Connection` object. For testing purposes, one can easily create a mock object of the configuration and inject that into the `Connection` object

- Symfony and Zend Framework 2 already have containers for DI that create objects via a configuration array and inject them where needed (i.e. in Controllers)

UML Diagram

Code

You can also find this code on GitHub

DatabaseConfiguration.php

```php
1   <?php
2
3   namespace DesignPatterns\Structural\DependencyInjection;
4
5   class DatabaseConfiguration
6   {
7       /**
8        * @var string
9        */
10      private $host;
11
12      /**
13       * @var int
14       */
15      private $port;
16
17      /**
18       * @var string
19       */
20      private $username;
21
22      /**
23       * @var string
24       */
25      private $password;
26
27      public function __construct(string $host, int $port, string $username, string
    ↪$password)
28      {
29          $this->host = $host;
30          $this->port = $port;
31          $this->username = $username;
32          $this->password = $password;
33      }
34
35      public function getHost(): string
36      {
37          return $this->host;
38      }
39
40      public function getPort(): int
41      {
42          return $this->port;
43      }
44
45      public function getUsername(): string
46      {
47          return $this->username;
48      }
49
50      public function getPassword(): string
51      {
52          return $this->password;
53      }
54  }
```

DatabaseConnection.php

```php
1   <?php
2
3   namespace DesignPatterns\Structural\DependencyInjection;
4
5   class DatabaseConnection
6   {
7       /**
8        * @var DatabaseConfiguration
9        */
10      private $configuration;
11
12      /**
13       * @param DatabaseConfiguration $config
14       */
15      public function __construct(DatabaseConfiguration $config)
16      {
17          $this->configuration = $config;
18      }
19
20      public function getDsn(): string
21      {
22          // this is just for the sake of demonstration, not a real DSN
23          // notice that only the injected config is used here, so there is
24          // a real separation of concerns here
25
26          return sprintf(
27              '%s:%s@%s:%d',
28              $this->configuration->getUsername(),
29              $this->configuration->getPassword(),
30              $this->configuration->getHost(),
31              $this->configuration->getPort()
32          );
33      }
34  }
```

Test

Tests/DependencyInjectionTest.php

```php
1   <?php
2
3   namespace DesignPatterns\Structural\DependencyInjection\Tests;
4
5   use DesignPatterns\Structural\DependencyInjection\DatabaseConfiguration;
6   use DesignPatterns\Structural\DependencyInjection\DatabaseConnection;
7   use PHPUnit\Framework\TestCase;
8
9   class DependencyInjectionTest extends TestCase
10  {
11      public function testDependencyInjection()
12      {
13          $config = new DatabaseConfiguration('localhost', 3306, 'domnikl', '1234');
14          $connection = new DatabaseConnection($config);
15
16          $this->assertEquals('domnikl:1234@localhost:3306', $connection->getDsn());
17      }
```

(continues on next page)

```
18    }
```

1.2.7 Facade

Purpose

The primary goal of a Facade Pattern is not to avoid you having to read the manual of a complex API. It's only a side-effect. The first goal is to reduce coupling and follow the Law of Demeter.
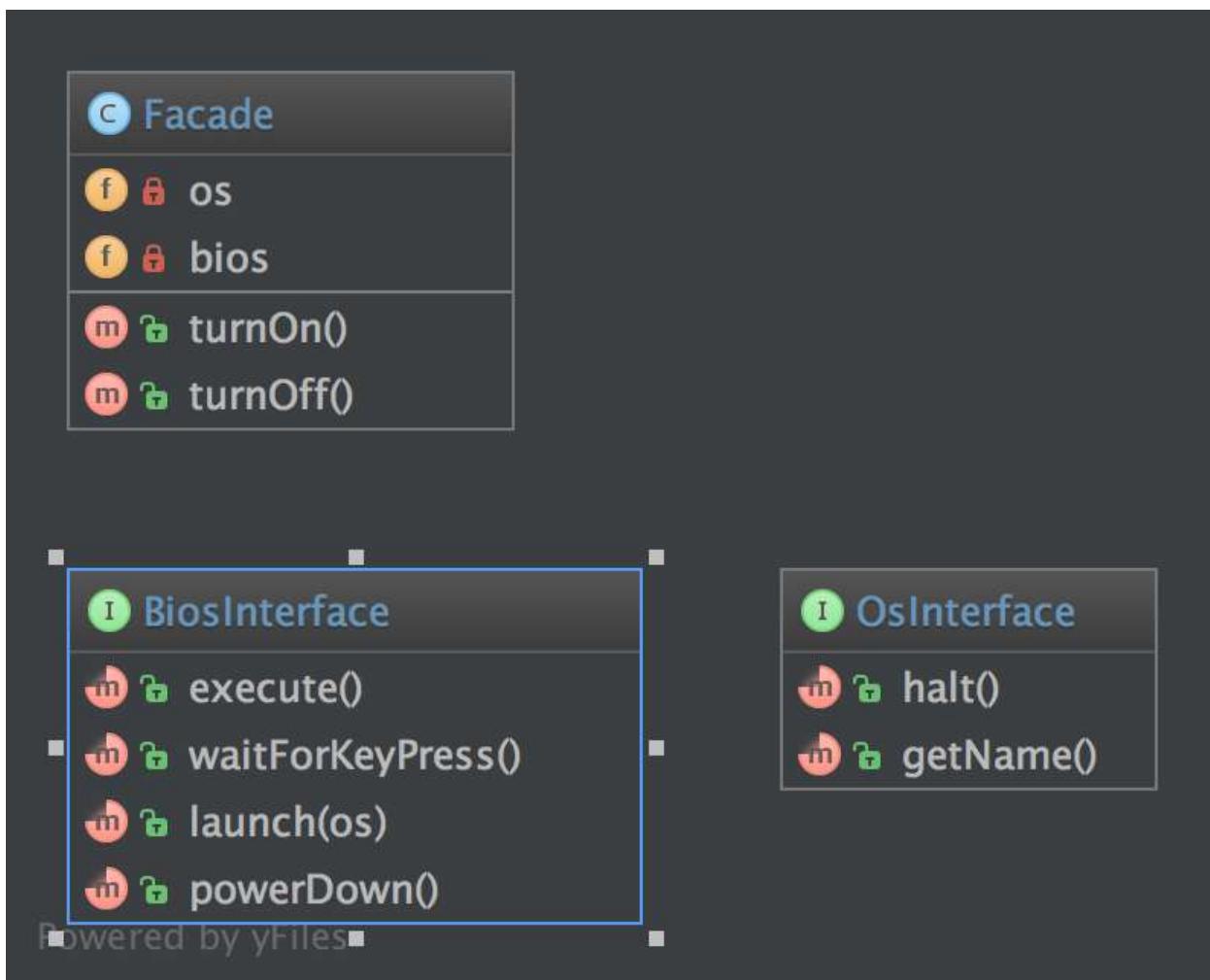
A Facade is meant to decouple a client and a sub-system by embedding many (but sometimes just one) interface, and of course to reduce complexity.

- A facade does not forbid you the access to the sub-system
- You can (you should) have multiple facades for one sub-system

That's why a good facade has no `new` in it. If there are multiple creations for each method, it is not a Facade, it's a Builder or a [Abstract|Static|Simple] Factory [Method].

The best facade has no `new` and a constructor with interface-type-hinted parameters. If you need creation of new instances, use a Factory as argument.

UML Diagram

Code

You can also find this code on GitHub

Facade.php

```php
<?php

namespace DesignPatterns\Structural\Facade;

class Facade
{
    /**
     * @var OsInterface
     */
    private $os;

    /**
     * @var BiosInterface
```

(continues on next page)

```php
14          */
15         private $bios;
16
17         /**
18          * @param BiosInterface $bios
19          * @param OsInterface   $os
20          */
21         public function __construct(BiosInterface $bios, OsInterface $os)
22         {
23             $this->bios = $bios;
24             $this->os = $os;
25         }
26
27         public function turnOn()
28         {
29             $this->bios->execute();
30             $this->bios->waitForKeyPress();
31             $this->bios->launch($this->os);
32         }
33
34         public function turnOff()
35         {
36             $this->os->halt();
37             $this->bios->powerDown();
38         }
39     }
```

OsInterface.php

```php
1   <?php
2
3   namespace DesignPatterns\Structural\Facade;
4
5   interface OsInterface
6   {
7       public function halt();
8
9       public function getName(): string;
10  }
```

BiosInterface.php

```php
1   <?php
2
3   namespace DesignPatterns\Structural\Facade;
4
5   interface BiosInterface
6   {
7       public function execute();
8
9       public function waitForKeyPress();
10
11      public function launch(OsInterface $os);
12
13      public function powerDown();
14  }
```

Test

Tests/FacadeTest.php

```php
<?php

namespace DesignPatterns\Structural\Facade\Tests;

use DesignPatterns\Structural\Facade\Facade;
use DesignPatterns\Structural\Facade\OsInterface;
use PHPUnit\Framework\TestCase;

class FacadeTest extends TestCase
{
    public function testComputerOn()
    {
        /** @var OsInterface|\PHPUnit_Framework_MockObject_MockObject $os */
        $os = $this->createMock('DesignPatterns\Structural\Facade\OsInterface');

        $os->method('getName')
            ->will($this->returnValue('Linux'));

        $bios = $this->getMockBuilder('DesignPatterns\Structural\Facade\BiosInterface')
            ->setMethods(['launch', 'execute', 'waitForKeyPress'])
            ->disableAutoload()
            ->getMock();

        $bios->expects($this->once())
            ->method('launch')
            ->with($os);

        $facade = new Facade($bios, $os);

        // the facade interface is simple
        $facade->turnOn();

        // but you can also access the underlying components
        $this->assertEquals('Linux', $os->getName());
    }
}
```

1.2.8 Fluent Interface
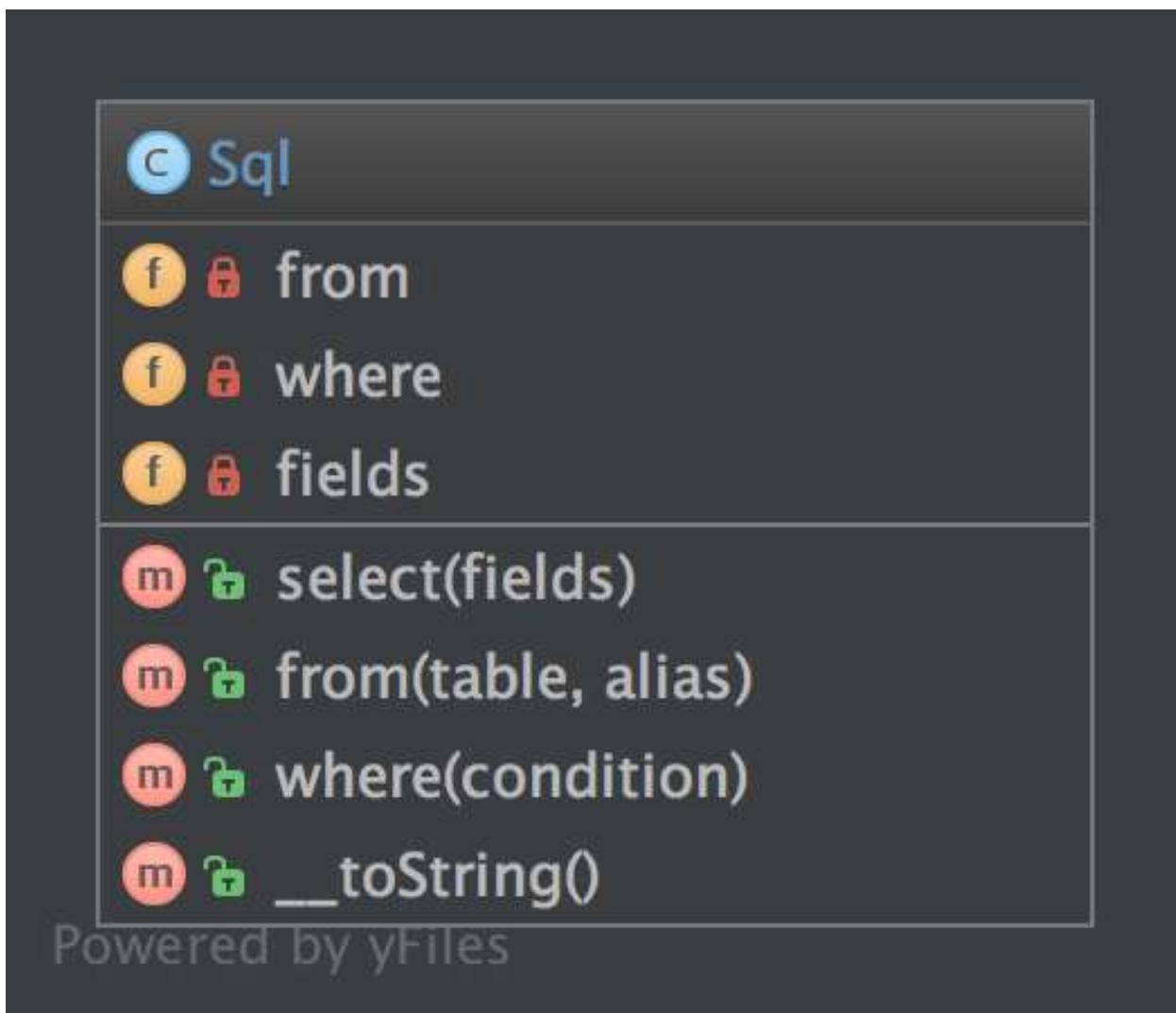
Purpose

To write code that is easy readable just like sentences in a natural language (like English).

Examples

- Doctrine2's QueryBuilder works something like that example class below
- PHPUnit uses fluent interfaces to build mock objects
- Yii Framework: CDbCommand and CActiveRecord use this pattern, too

UML Diagram

Code

You can also find this code on GitHub

Sql.php

```php
<?php

namespace DesignPatterns\Structural\FluentInterface;

class Sql
{
    /**
     * @var array
     */
    private $fields = [];

```

(continues on next page)

```php
12       /**
13        * @var array
14        */
15       private $from = [];
16
17       /**
18        * @var array
19        */
20       private $where = [];
21
22       public function select(array $fields): Sql
23       {
24           $this->fields = $fields;
25
26           return $this;
27       }
28
29       public function from(string $table, string $alias): Sql
30       {
31           $this->from[] = $table.' AS '.$alias;
32
33           return $this;
34       }
35
36       public function where(string $condition): Sql
37       {
38           $this->where[] = $condition;
39
40           return $this;
41       }
42
43       public function __toString(): string
44       {
45           return sprintf(
46               'SELECT %s FROM %s WHERE %s',
47               join(', ', $this->fields),
48               join(', ', $this->from),
49               join(' AND ', $this->where)
50           );
51       }
52   }
```

Test

Tests/FluentInterfaceTest.php

```php
1   <?php
2
3   namespace DesignPatterns\Structural\FluentInterface\Tests;
4
5   use DesignPatterns\Structural\FluentInterface\Sql;
6   use PHPUnit\Framework\TestCase;
7
8   class FluentInterfaceTest extends TestCase
9   {
```

```
10      public function testBuildSQL()
11      {
12          $query = (new Sql())
13                  ->select(['foo', 'bar'])
14                  ->from('foobar', 'f')
15                  ->where('f.bar = ?');
16
17          $this->assertEquals('SELECT foo, bar FROM foobar AS f WHERE f.bar = ?',␣
    ↪(string) $query);
18      }
19  }
```
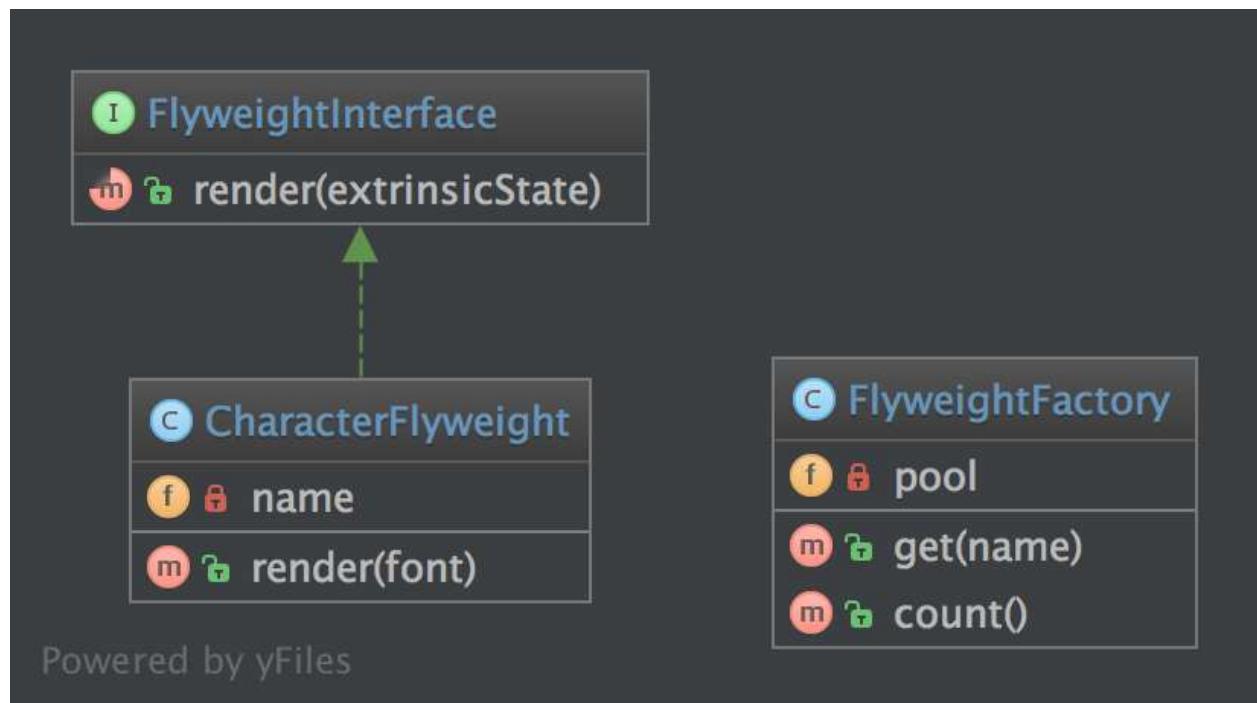
1.2.9 Flyweight

#### Purpose

To minimise memory usage, a Flyweight shares as much as possible memory with similar objects. It is needed when a large amount of objects is used that don't differ much in state. A common practice is to hold state in external data structures and pass them to the flyweight object when needed.

UML Diagram

Code

You can also find this code on GitHub

FlyweightInterface.php

```php
1   <?php
2
3   namespace DesignPatterns\Structural\Flyweight;
4
5   interface FlyweightInterface
6   {
7       public function render(string $extrinsicState): string;
8   }
```

CharacterFlyweight.php

```php
1   <?php
2
3   namespace DesignPatterns\Structural\Flyweight;
4
5   /**
6    * Implements the flyweight interface and adds storage for intrinsic state, if any.
7    * Instances of concrete flyweights are shared by means of a factory.
8    */
9   class CharacterFlyweight implements FlyweightInterface
10  {
11      /**
12       * Any state stored by the concrete flyweight must be independent of its context.
13       * For flyweights representing characters, this is usually the corresponding␣
    ↪character code.
14       *
15       * @var string
16       */
17      private $name;
18
19      public function __construct(string $name)
20      {
21          $this->name = $name;
22      }
23
24      public function render(string $font): string
25      {
26          // Clients supply the context-dependent information that the flyweight needs␣
    ↪to draw itself
27          // For flyweights representing characters, extrinsic state usually contains␣
    ↪e.g. the font.
28
29          return sprintf('Character %s with font %s', $this->name, $font);
30      }
31  }
```

FlyweightFactory.php

```php
1   <?php
2
3   namespace DesignPatterns\Structural\Flyweight;
4
5   /**
6    * A factory manages shared flyweights. Clients should not instantiate them directly,
7    * but let the factory take care of returning existing objects or creating new ones.
8    */
9   class FlyweightFactory implements \Countable
```

```php
10  {
11      /**
12       * @var CharacterFlyweight[]
13       */
14      private $pool = [];
15
16      public function get(string $name): CharacterFlyweight
17      {
18          if (!isset($this->pool[$name])) {
19              $this->pool[$name] = new CharacterFlyweight($name);
20          }
21
22          return $this->pool[$name];
23      }
24
25      public function count(): int
26      {
27          return count($this->pool);
28      }
29  }
```

Test

Tests/FlyweightTest.php

```php
1   <?php
2
3   namespace DesignPatterns\Structural\Flyweight\Tests;
4
5   use DesignPatterns\Structural\Flyweight\FlyweightFactory;
6   use PHPUnit\Framework\TestCase;
7
8   class FlyweightTest extends TestCase
9   {
10      private $characters = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k',
11          'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z'];
12      private $fonts = ['Arial', 'Times New Roman', 'Verdana', 'Helvetica'];
13
14      public function testFlyweight()
15      {
16          $factory = new FlyweightFactory();
17
18          foreach ($this->characters as $char) {
19              foreach ($this->fonts as $font) {
20                  $flyweight = $factory->get($char);
21                  $rendered = $flyweight->render($font);
22
23                  $this->assertEquals(sprintf('Character %s with font %s', $char,
    ↪$font), $rendered);
24              }
25          }
26
27          // Flyweight pattern ensures that instances are shared
28          // instead of having hundreds of thousands of individual objects
29          // there must be one instance for every char that has been reused for
    ↪displaying in different fonts
```

```
30          $this->assertCount(count($this->characters), $factory);
31      }
32  }
```

1.2.10 Proxy

Purpose
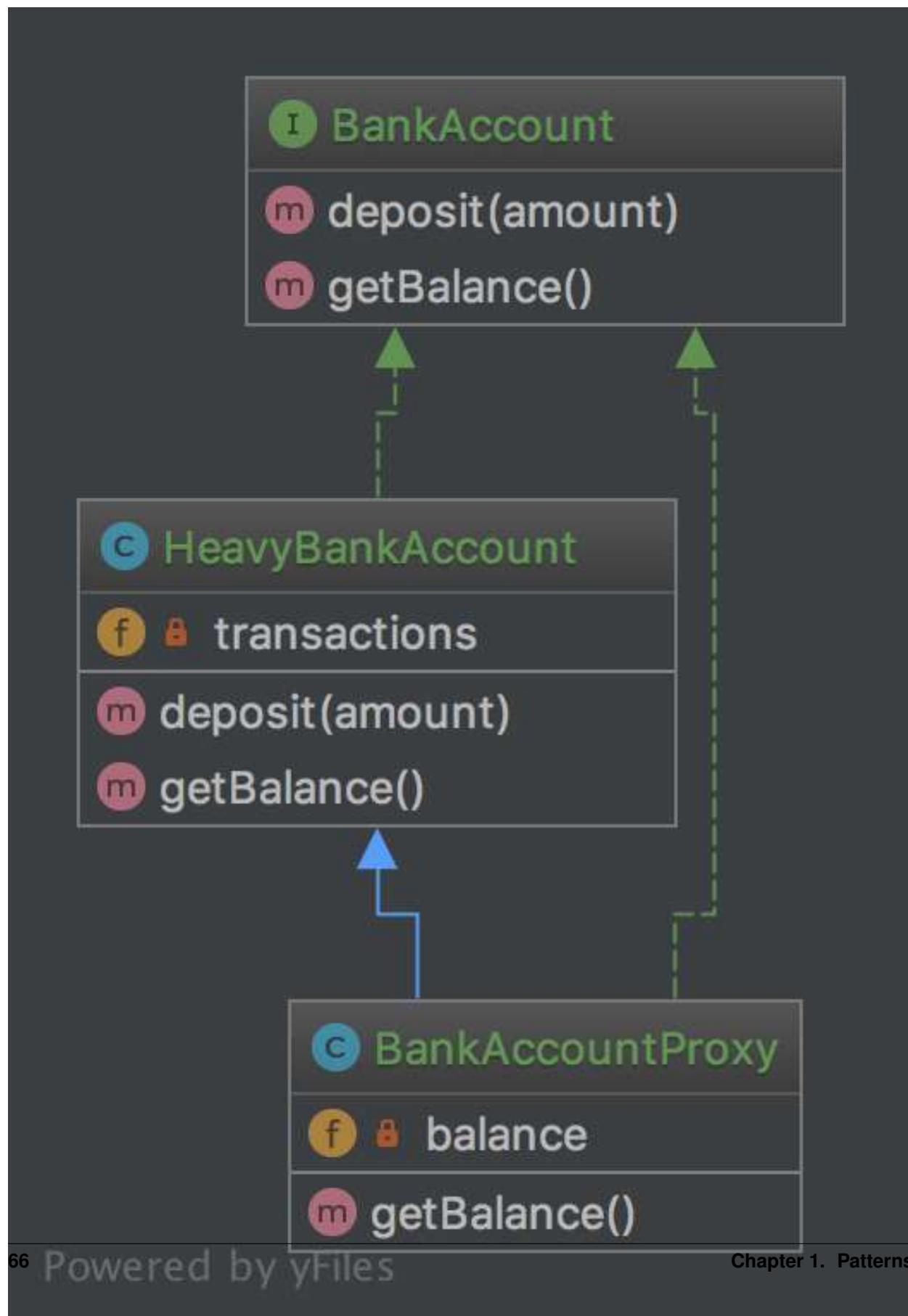
To interface to anything that is expensive or impossible to duplicate.

Examples

- Doctrine2 uses proxies to implement framework magic (e.g. lazy initialization) in them, while the user still works with his own entity classes and will never use nor touch the proxies

UML Diagram

Powered by yFiles

Code

You can also find this code on GitHub

BankAccount.php

```php
<?php

namespace DesignPatterns\Structural\Proxy;

interface BankAccount
{
    public function deposit(int $amount);

    public function getBalance(): int;
}
```

HeavyBankAccount.php

```php
<?php

namespace DesignPatterns\Structural\Proxy;

class HeavyBankAccount implements BankAccount
{
    /**
     * @var int[]
     */
    private $transactions = [];

    public function deposit(int $amount)
    {
        $this->transactions[] = $amount;
    }

    public function getBalance(): int
    {
        // this is the heavy part, imagine all the transactions even from
        // years and decades ago must be fetched from a database or web service
        // and the balance must be calculated from it

        return array_sum($this->transactions);
    }
}
```

BankAccountProxy.php

```php
<?php

namespace DesignPatterns\Structural\Proxy;

class BankAccountProxy extends HeavyBankAccount implements BankAccount
{
    /**
     * @var int
     */
    private $balance;

```

(continues on next page)

```
12      public function getBalance(): int
13      {
14          // because calculating balance is so expensive,
15          // the usage of BankAccount::getBalance() is delayed until it really is needed
16          // and will not be calculated again for this instance
17
18          if ($this->balance === null) {
19              $this->balance = parent::getBalance();
20          }
21
22          return $this->balance;
23      }
24  }
```

Test

1.2.11 Registry
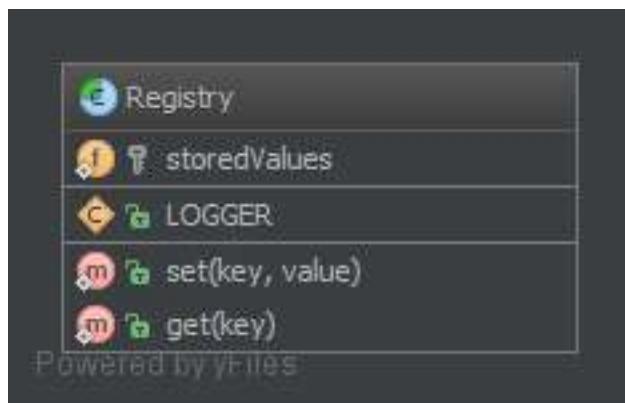
Purpose

To implement a central storage for objects often used throughout the application, is typically implemented using an abstract class with only static methods (or using the Singleton pattern). Remember that this introduces global state, which should be avoided at all times! Instead implement it using Dependency Injection!

Examples

- Zend Framework 1: `Zend_Registry` holds the application's logger object, front controller etc.

- Yii Framework: `CWebApplication` holds all the application components, such as `CWebUser`, `CUrlManager`, etc.

UML Diagram

Code

You can also find this code on GitHub

Registry.php

```php
<?php

namespace DesignPatterns\Structural\Registry;

abstract class Registry
{
    const LOGGER = 'logger';

    /**
     * this introduces global state in your application which can not be mocked up
     ↪for testing
     * and is therefor considered an anti-pattern! Use dependency injection instead!
     *
     * @var array
     */
    private static $storedValues = [];

    /**
     * @var array
     */
    private static $allowedKeys = [
        self::LOGGER,
    ];

    /**
     * @param string $key
     * @param mixed  $value
     *
     * @return void
     */
    public static function set(string $key, $value)
    {
        if (!in_array($key, self::$allowedKeys)) {
            throw new \InvalidArgumentException('Invalid key given');
        }

        self::$storedValues[$key] = $value;
    }

    /**
     * @param string $key
     *
     * @return mixed
     */
    public static function get(string $key)
    {
        if (!in_array($key, self::$allowedKeys) || !isset(self::$storedValues[$key]))
     ↪{
            throw new \InvalidArgumentException('Invalid key given');
        }

        return self::$storedValues[$key];
    }
}
```

Test

Tests/RegistryTest.php

```php
<?php

namespace DesignPatterns\Structural\Registry\Tests;

use DesignPatterns\Structural\Registry\Registry;
use stdClass;
use PHPUnit\Framework\TestCase;

class RegistryTest extends TestCase
{
    public function testSetAndGetLogger()
    {
        $key = Registry::LOGGER;
        $logger = new stdClass();

        Registry::set($key, $logger);
        $storedLogger = Registry::get($key);

        $this->assertSame($logger, $storedLogger);
        $this->assertInstanceOf(stdClass::class, $storedLogger);
    }

    /**
     * @expectedException \InvalidArgumentException
     */
    public function testThrowsExceptionWhenTryingToSetInvalidKey()
    {
        Registry::set('foobar', new stdClass());
    }

    /**
     * notice @runInSeparateProcess here: without it, a previous test might have set
     ↪it already and
     * testing would not be possible. That's why you should implement Dependency
     ↪Injection where an
     * injected class may easily be replaced by a mockup
     *
     * @runInSeparateProcess
     * @expectedException \InvalidArgumentException
     */
    public function testThrowsExceptionWhenTryingToGetNotSetKey()
    {
        Registry::get(Registry::LOGGER);
    }
}
```

1.3 Behavioral

In software engineering, behavioral design patterns are design patterns that identify common communication patterns between objects and realize these patterns. By doing so, these patterns increase flexibility in carrying out this communication.

1.3.1 Chain Of Responsibilities
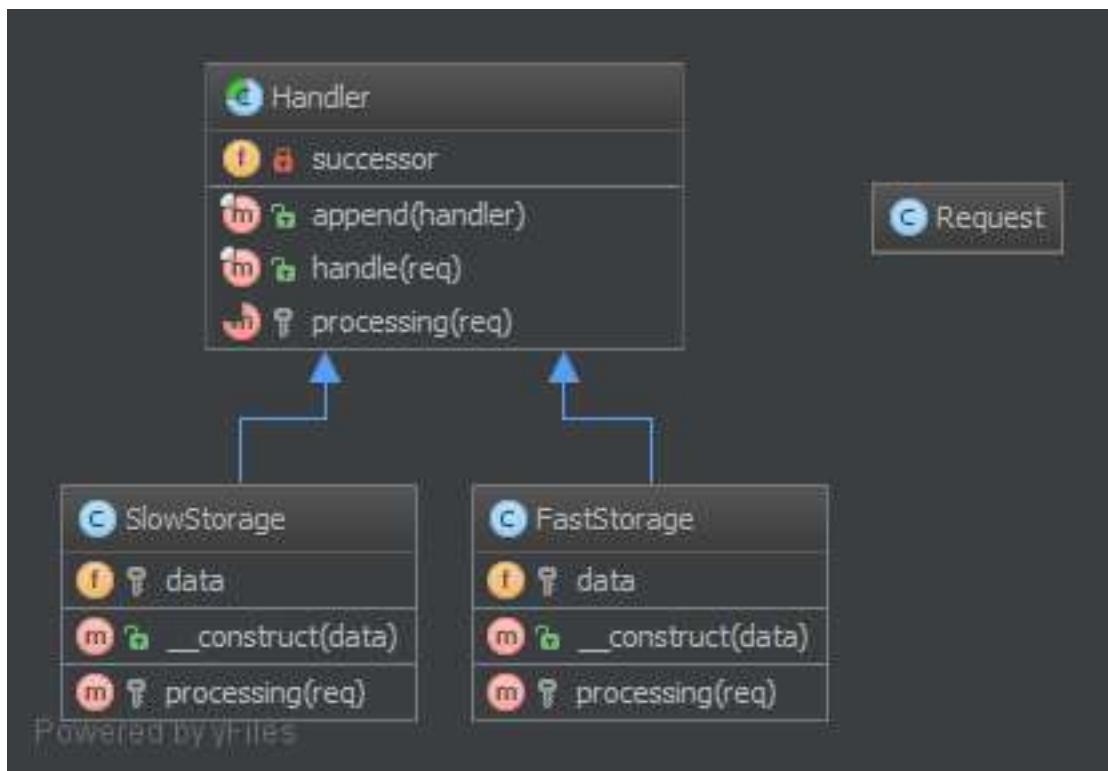
Purpose

To build a chain of objects to handle a call in sequential order. If one object cannot handle a call, it delegates the call to the next in the chain and so forth.

Examples

- logging framework, where each chain element decides autonomously what to do with a log message
- a Spam filter
- Caching: first object is an instance of e.g. a Memcached Interface, if that "misses" it delegates the call to the database interface
- Yii Framework: CFilterChain is a chain of controller action filters. the executing point is passed from one filter to the next along the chain, and only if all filters say "yes", the action can be invoked at last.

UML Diagram

Code

You can also find this code on GitHub

Handler.php

```php
1  <?php
2
3  namespace DesignPatterns\Behavioral\ChainOfResponsibilities;
4
5  use Psr\Http\Message\RequestInterface;
6  use Psr\Http\Message\ResponseInterface;
7
8  abstract class Handler
9  {
10     /**
11      * @var Handler|null
12      */
13     private $successor = null;
14
15     public function __construct(Handler $handler = null)
16     {
17         $this->successor = $handler;
18     }
19
20     /**
21      * This approach by using a template method pattern ensures you that
22      * each subclass will not forget to call the successor
23      *
24      * @param RequestInterface $request
25      *
26      * @return string|null
27      */
28     final public function handle(RequestInterface $request)
29     {
30         $processed = $this->processing($request);
31
32         if ($processed === null) {
33             // the request has not been processed by this handler => see the next
34             if ($this->successor !== null) {
35                 $processed = $this->successor->handle($request);
36             }
37         }
38
39         return $processed;
40     }
41
42     abstract protected function processing(RequestInterface $request);
43  }
```

Responsible/FastStorage.php

```php
1  <?php
2
3  namespace DesignPatterns\Behavioral\ChainOfResponsibilities\Responsible;
4
5  use DesignPatterns\Behavioral\ChainOfResponsibilities\Handler;
6  use Psr\Http\Message\RequestInterface;
7
8  class HttpInMemoryCacheHandler extends Handler
9  {
10     /**
11      * @var array
```

(continues on next page)

```php
12          */
13         private $data;
14
15         /**
16          * @param array $data
17          * @param Handler|null $successor
18          */
19         public function __construct(array $data, Handler $successor = null)
20         {
21             parent::__construct($successor);
22
23             $this->data = $data;
24         }
25
26         /**
27          * @param RequestInterface $request
28          *
29          * @return string|null
30          */
31         protected function processing(RequestInterface $request)
32         {
33             $key = sprintf(
34                 '%s?%s',
35                 $request->getUri()->getPath(),
36                 $request->getUri()->getQuery()
37             );
38
39             if ($request->getMethod() == 'GET' && isset($this->data[$key])) {
40                 return $this->data[$key];
41             }
42
43             return null;
44         }
45     }
```

Responsible/SlowStorage.php

```php
1   <?php
2
3   namespace DesignPatterns\Behavioral\ChainOfResponsibilities\Responsible;
4
5   use DesignPatterns\Behavioral\ChainOfResponsibilities\Handler;
6   use Psr\Http\Message\RequestInterface;
7
8   class SlowDatabaseHandler extends Handler
9   {
10      /**
11       * @param RequestInterface $request
12       *
13       * @return string|null
14       */
15      protected function processing(RequestInterface $request)
16      {
17          // this is a mockup, in production code you would ask a slow (compared to in-
    →memory) DB for the results
18
19          return 'Hello World!';
```

```
20        }
21    }
```

Test

Tests/ChainTest.php

```php
1   <?php
2
3   namespace DesignPatterns\Behavioral\ChainOfResponsibilities\Tests;
4
5   use DesignPatterns\Behavioral\ChainOfResponsibilities\Handler;
6   use
    →DesignPatterns\Behavioral\ChainOfResponsibilities\Responsible\HttpInMemoryCacheHandler;
    →
7   use DesignPatterns\Behavioral\ChainOfResponsibilities\Responsible\SlowDatabaseHandler;
8   use PHPUnit\Framework\TestCase;
9
10  class ChainTest extends TestCase
11  {
12      /**
13       * @var Handler
14       */
15      private $chain;
16
17      protected function setUp()
18      {
19          $this->chain = new HttpInMemoryCacheHandler(
20              ['/foo/bar?index=1' => 'Hello In Memory!'],
21              new SlowDatabaseHandler()
22          );
23      }
24
25      public function testCanRequestKeyInFastStorage()
26      {
27          $uri = $this->createMock('Psr\Http\Message\UriInterface');
28          $uri->method('getPath')->willReturn('/foo/bar');
29          $uri->method('getQuery')->willReturn('index=1');
30
31          $request = $this->createMock('Psr\Http\Message\RequestInterface');
32          $request->method('getMethod')
33              ->willReturn('GET');
34          $request->method('getUri')->willReturn($uri);
35
36          $this->assertEquals('Hello In Memory!', $this->chain->handle($request));
37      }
38
39      public function testCanRequestKeyInSlowStorage()
40      {
41          $uri = $this->createMock('Psr\Http\Message\UriInterface');
42          $uri->method('getPath')->willReturn('/foo/baz');
43          $uri->method('getQuery')->willReturn('');
44
45          $request = $this->createMock('Psr\Http\Message\RequestInterface');
46          $request->method('getMethod')
```

```
47              ->willReturn('GET');
48          $request->method('getUri')->willReturn($uri);
49
50          $this->assertEquals('Hello World!', $this->chain->handle($request));
51      }
52  }
```

1.3.2 Command

Purpose

To encapsulate invocation and decoupling.
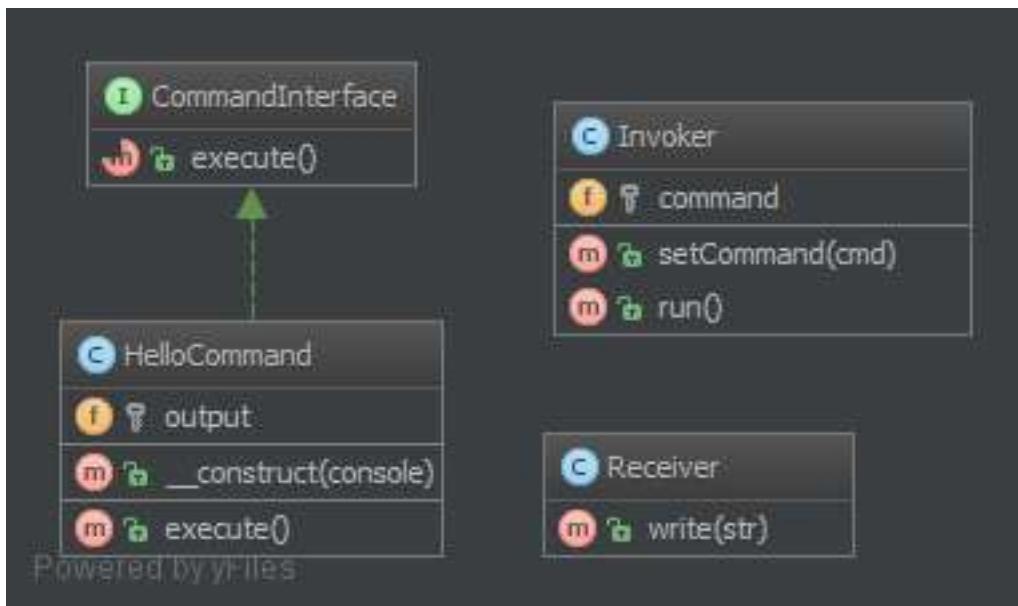
We have an Invoker and a Receiver. This pattern uses a "Command" to delegate the method call against the Receiver and presents the same method "execute". Therefore, the Invoker just knows to call "execute" to process the Command of the client. The Receiver is decoupled from the Invoker.

The second aspect of this pattern is the undo(), which undoes the method execute(). Command can also be aggregated to combine more complex commands with minimum copy-paste and relying on composition over inheritance.

Examples

- A text editor : all events are Command which can be undone, stacked and saved.

- Symfony2: SF2 Commands that can be run from the CLI are built with just the Command pattern in mind

- big CLI tools use subcommands to distribute various tasks and pack them in "modules", each of these can be implemented with the Command pattern (e.g. vagrant)

UML Diagram

Code

You can also find this code on GitHub

CommandInterface.php

```php
<?php

namespace DesignPatterns\Behavioral\Command;

interface CommandInterface
{
    /**
     * this is the most important method in the Command pattern,
     * The Receiver goes in the constructor.
     */
    public function execute();
}
```

HelloCommand.php

```php
<?php

namespace DesignPatterns\Behavioral\Command;

/**
 * This concrete command calls "print" on the Receiver, but an external
 * invoker just knows that it can call "execute"
 */
class HelloCommand implements CommandInterface
{
    /**
     * @var Receiver
     */
    private $output;

    /**
     * Each concrete command is built with different receivers.
     * There can be one, many or completely no receivers, but there can be other
     commands in the parameters
     *
     * @param Receiver $console
     */
    public function __construct(Receiver $console)
    {
        $this->output = $console;
    }

    /**
     * execute and output "Hello World".
     */
    public function execute()
    {
        // sometimes, there is no receiver and this is the command which does all the
     work
        $this->output->write('Hello World');
    }
}
```

Receiver.php

```php
<?php

namespace DesignPatterns\Behavioral\Command;

/**
 * Receiver is specific service with its own contract and can be only concrete.
 */
class Receiver
{
    /**
     * @var bool
     */
    private $enableDate = false;

    /**
     * @var string[]
     */
    private $output = [];

    /**
     * @param string $str
     */
    public function write(string $str)
    {
        if ($this->enableDate) {
            $str .= ' ['.date('Y-m-d').']';
        }

        $this->output[] = $str;
    }

    public function getOutput(): string
    {
        return join("\n", $this->output);
    }

    /**
     * Enable receiver to display message date
     */
    public function enableDate()
    {
        $this->enableDate = true;
    }

    /**
     * Disable receiver to display message date
     */
    public function disableDate()
    {
        $this->enableDate = false;
    }
}
```

Invoker.php

```php
<?php
```

(continues on next page)

```php
namespace DesignPatterns\Behavioral\Command;

/**
 * Invoker is using the command given to it.
 * Example : an Application in SF2.
 */
class Invoker
{
    /**
     * @var CommandInterface
     */
    private $command;

    /**
     * in the invoker we find this kind of method for subscribing the command
     * There can be also a stack, a list, a fixed set ...
     *
     * @param CommandInterface $cmd
     */
    public function setCommand(CommandInterface $cmd)
    {
        $this->command = $cmd;
    }

    /**
     * executes the command; the invoker is the same whatever is the command
     */
    public function run()
    {
        $this->command->execute();
    }
}
```

Test

Tests/CommandTest.php

```php
<?php

namespace DesignPatterns\Behavioral\Command\Tests;

use DesignPatterns\Behavioral\Command\HelloCommand;
use DesignPatterns\Behavioral\Command\Invoker;
use DesignPatterns\Behavioral\Command\Receiver;
use PHPUnit\Framework\TestCase;

class CommandTest extends TestCase
{
    public function testInvocation()
    {
        $invoker = new Invoker();
        $receiver = new Receiver();

        $invoker->setCommand(new HelloCommand($receiver));
```

```
18          $invoker->run();
19          $this->assertEquals('Hello World', $receiver->getOutput());
20      }
21  }
```

1.3.3 Iterator

Purpose

To make an object iterable and to make it appear like a collection of objects.
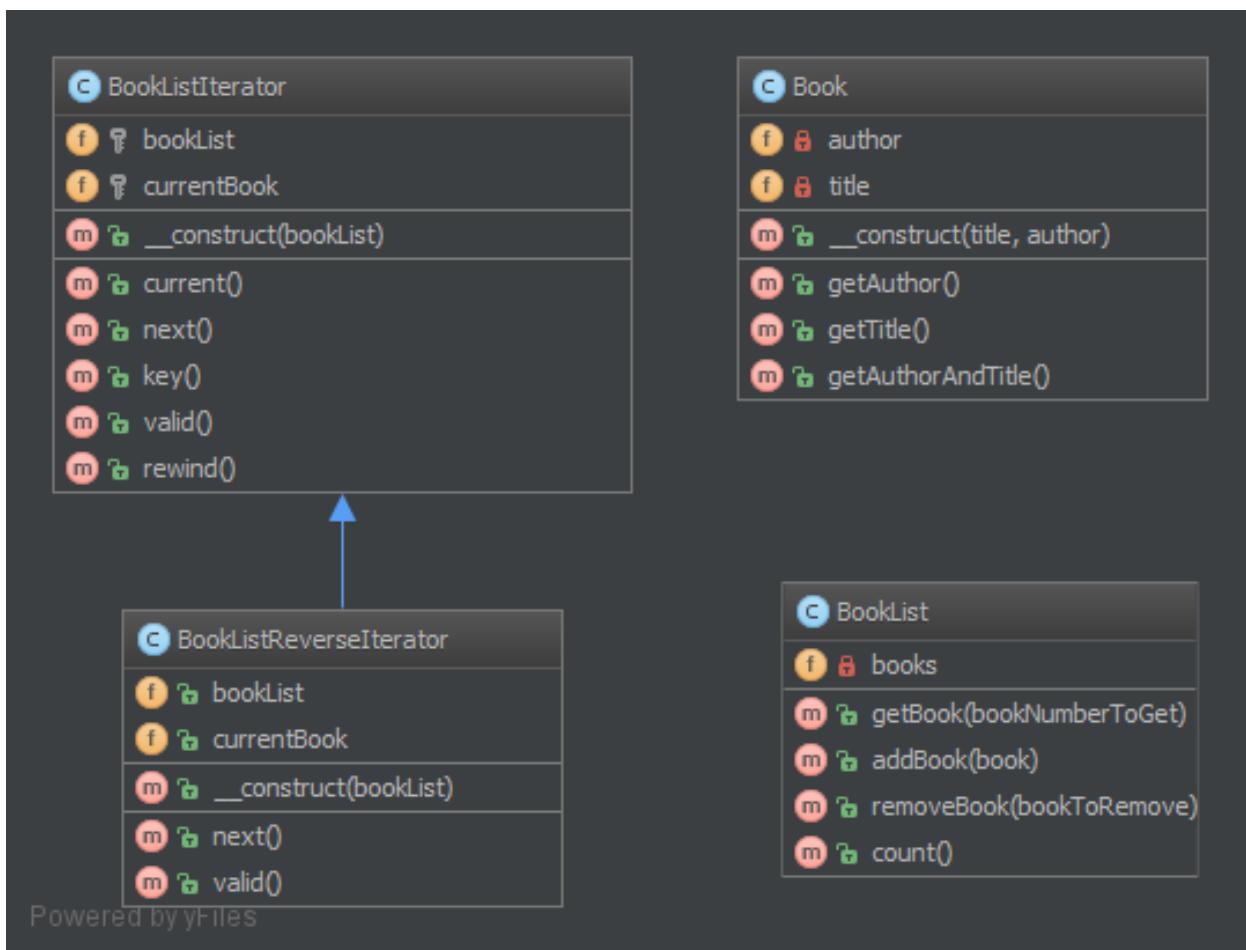
Examples

- to process a file line by line by just running over all lines (which have an object representation) for a file (which of course is an object, too)

Note

Standard PHP Library (SPL) defines an interface Iterator which is best suited for this! Often you would want to implement the Countable interface too, to allow `count($object)` on your iterable object

UML Diagram

Code

You can also find this code on GitHub

Book.php

```php
<?php

namespace DesignPatterns\Behavioral\Iterator;

class Book
{
    /**
     * @var string
     */
    private $author;

    /**
     * @var string
     */
    private $title;
```

```php
16
17      public function __construct(string $title, string $author)
18      {
19          $this->author = $author;
20          $this->title = $title;
21      }
22
23      public function getAuthor(): string
24      {
25          return $this->author;
26      }
27
28      public function getTitle(): string
29      {
30          return $this->title;
31      }
32
33      public function getAuthorAndTitle(): string
34      {
35          return $this->getTitle().' by '.$this->getAuthor();
36      }
37  }
```

BookList.php

```php
1   <?php
2
3   namespace DesignPatterns\Behavioral\Iterator;
4
5   class BookList implements \Countable, \Iterator
6   {
7       /**
8        * @var Book[]
9        */
10      private $books = [];
11
12      /**
13       * @var int
14       */
15      private $currentIndex = 0;
16
17      public function addBook(Book $book)
18      {
19          $this->books[] = $book;
20      }
21
22      public function removeBook(Book $bookToRemove)
23      {
24          foreach ($this->books as $key => $book) {
25              if ($book->getAuthorAndTitle() === $bookToRemove->getAuthorAndTitle()) {
26                  unset($this->books[$key]);
27              }
28          }
29
30          $this->books = array_values($this->books);
31      }
32
```

```php
33      public function count(): int
34      {
35          return count($this->books);
36      }
37
38      public function current(): Book
39      {
40          return $this->books[$this->currentIndex];
41      }
42
43      public function key(): int
44      {
45          return $this->currentIndex;
46      }
47
48      public function next()
49      {
50          $this->currentIndex++;
51      }
52
53      public function rewind()
54      {
55          $this->currentIndex = 0;
56      }
57
58      public function valid(): bool
59      {
60          return isset($this->books[$this->currentIndex]);
61      }
62  }
```

Test

Tests/IteratorTest.php

```php
1   <?php
2
3   namespace DesignPatterns\Behavioral\Iterator\Tests;
4
5   use DesignPatterns\Behavioral\Iterator\Book;
6   use DesignPatterns\Behavioral\Iterator\BookList;
7   use DesignPatterns\Behavioral\Iterator\BookListIterator;
8   use DesignPatterns\Behavioral\Iterator\BookListReverseIterator;
9   use PHPUnit\Framework\TestCase;
10
11  class IteratorTest extends TestCase
12  {
13      public function testCanIterateOverBookList()
14      {
15          $bookList = new BookList();
16          $bookList->addBook(new Book('Learning PHP Design Patterns', 'William Sanders
    ↪'));
17          $bookList->addBook(new Book('Professional Php Design Patterns', 'Aaron Saray
    ↪'));
18          $bookList->addBook(new Book('Clean Code', 'Robert C. Martin'));
```

```php
19
20          $books = [];
21
22          foreach ($bookList as $book) {
23              $books[] = $book->getAuthorAndTitle();
24          }
25
26          $this->assertEquals(
27              [
28                  'Learning PHP Design Patterns by William Sanders',
29                  'Professional Php Design Patterns by Aaron Saray',
30                  'Clean Code by Robert C. Martin',
31              ],
32              $books
33          );
34      }
35
36      public function testCanIterateOverBookListAfterRemovingBook()
37      {
38          $book = new Book('Clean Code', 'Robert C. Martin');
39          $book2 = new Book('Professional Php Design Patterns', 'Aaron Saray');
40
41          $bookList = new BookList();
42          $bookList->addBook($book);
43          $bookList->addBook($book2);
44          $bookList->removeBook($book);
45
46          $books = [];
47          foreach ($bookList as $book) {
48              $books[] = $book->getAuthorAndTitle();
49          }
50
51          $this->assertEquals(
52              ['Professional Php Design Patterns by Aaron Saray'],
53              $books
54          );
55      }
56
57      public function testCanAddBookToList()
58      {
59          $book = new Book('Clean Code', 'Robert C. Martin');
60
61          $bookList = new BookList();
62          $bookList->addBook($book);
63
64          $this->assertCount(1, $bookList);
65      }
66
67      public function testCanRemoveBookFromList()
68      {
69          $book = new Book('Clean Code', 'Robert C. Martin');
70
71          $bookList = new BookList();
72          $bookList->addBook($book);
73          $bookList->removeBook($book);
74
75          $this->assertCount(0, $bookList);
```

```
76        }
77    }
```
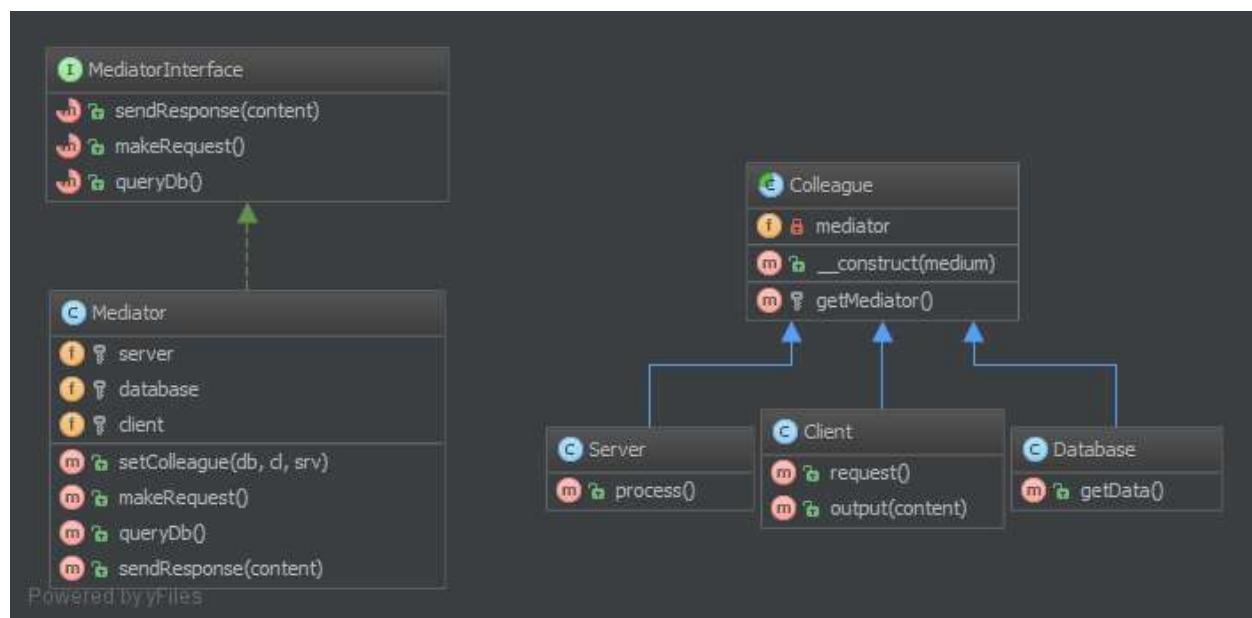
1.3.4 Mediator

Purpose

This pattern provides an easy way to decouple many components working together. It is a good alternative to Observer IF you have a "central intelligence", like a controller (but not in the sense of the MVC).

All components (called Colleague) are only coupled to the MediatorInterface and it is a good thing because in OOP, one good friend is better than many. This is the key-feature of this pattern.

UML Diagram

Code

You can also find this code on GitHub

MediatorInterface.php

```php
1   <?php
2
3   namespace DesignPatterns\Behavioral\Mediator;
4
5   /**
6    * MediatorInterface is a contract for the Mediator
7    * This interface is not mandatory but it is better for Liskov substitution principle
    ↪concerns.
8    */
```

```php
 9  interface MediatorInterface
10  {
11      /**
12       * sends the response.
13       *
14       * @param string $content
15       */
16      public function sendResponse($content);
17
18      /**
19       * makes a request
20       */
21      public function makeRequest();
22
23      /**
24       * queries the DB
25       */
26      public function queryDb();
27  }
```

Mediator.php

```php
 1  <?php
 2
 3  namespace DesignPatterns\Behavioral\Mediator;
 4
 5  /**
 6   * Mediator is the concrete Mediator for this design pattern
 7   *
 8   * In this example, I have made a "Hello World" with the Mediator Pattern
 9   */
10  class Mediator implements MediatorInterface
11  {
12      /**
13       * @var Subsystem\Server
14       */
15      private $server;
16
17      /**
18       * @var Subsystem\Database
19       */
20      private $database;
21
22      /**
23       * @var Subsystem\Client
24       */
25      private $client;
26
27      /**
28       * @param Subsystem\Database $database
29       * @param Subsystem\Client $client
30       * @param Subsystem\Server $server
31       */
32      public function __construct(Subsystem\Database $database, Subsystem\Client
    ↪$client, Subsystem\Server $server)
33      {
34          $this->database = $database;
```

```php
35          $this->server = $server;
36          $this->client = $client;
37
38          $this->database->setMediator($this);
39          $this->server->setMediator($this);
40          $this->client->setMediator($this);
41      }
42
43      public function makeRequest()
44      {
45          $this->server->process();
46      }
47
48      public function queryDb(): string
49      {
50          return $this->database->getData();
51      }
52
53      /**
54       * @param string $content
55       */
56      public function sendResponse($content)
57      {
58          $this->client->output($content);
59      }
60  }
```

Colleague.php

```php
1   <?php
2
3   namespace DesignPatterns\Behavioral\Mediator;
4
5   /**
6    * Colleague is an abstract colleague who works together but he only knows
7    * the Mediator, not other colleagues
8    */
9   abstract class Colleague
10  {
11      /**
12       * this ensures no change in subclasses.
13       *
14       * @var MediatorInterface
15       */
16      protected $mediator;
17
18      /**
19       * @param MediatorInterface $mediator
20       */
21      public function setMediator(MediatorInterface $mediator)
22      {
23          $this->mediator = $mediator;
24      }
25  }
```

Subsystem/Client.php

```php
1  <?php
2
3  namespace DesignPatterns\Behavioral\Mediator\Subsystem;
4
5  use DesignPatterns\Behavioral\Mediator\Colleague;
6
7  /**
8   * Client is a client that makes requests and gets the response.
9   */
10 class Client extends Colleague
11 {
12     public function request()
13     {
14         $this->mediator->makeRequest();
15     }
16
17     public function output(string $content)
18     {
19         echo $content;
20     }
21 }
```

Subsystem/Database.php

```php
1  <?php
2
3  namespace DesignPatterns\Behavioral\Mediator\Subsystem;
4
5  use DesignPatterns\Behavioral\Mediator\Colleague;
6
7  class Database extends Colleague
8  {
9      public function getData(): string
10     {
11         return 'World';
12     }
13 }
```

Subsystem/Server.php

```php
1  <?php
2
3  namespace DesignPatterns\Behavioral\Mediator\Subsystem;
4
5  use DesignPatterns\Behavioral\Mediator\Colleague;
6
7  class Server extends Colleague
8  {
9      public function process()
10     {
11         $data = $this->mediator->queryDb();
12         $this->mediator->sendResponse(sprintf("Hello %s", $data));
13     }
14 }
```

Test

Tests/MediatorTest.php

```php
<?php

namespace DesignPatterns\Tests\Mediator\Tests;

use DesignPatterns\Behavioral\Mediator\Mediator;
use DesignPatterns\Behavioral\Mediator\Subsystem\Client;
use DesignPatterns\Behavioral\Mediator\Subsystem\Database;
use DesignPatterns\Behavioral\Mediator\Subsystem\Server;
use PHPUnit\Framework\TestCase;

class MediatorTest extends TestCase
{
    public function testOutputHelloWorld()
    {
        $client = new Client();
        new Mediator(new Database(), $client, new Server());

        $this->expectOutputString('Hello World');
        $client->request();
    }
}
```

1.3.5 Memento

Purpose

It provides the ability to restore an object to it's previous state (undo via rollback) or to gain access to state of the object, without revealing it's implementation (i.e., the object is not required to have a function to return the current state).

The memento pattern is implemented with three objects: the Originator, a Caretaker and a Memento.

Memento – an object that *contains a concrete unique snapshot of state* of any object or resource: string, number, array, an instance of class and so on. The uniqueness in this case does not imply the prohibition existence of similar states in different snapshots. That means the state can be extracted as the independent clone. Any object stored in the Memento should be *a full copy of the original object rather than a reference* to the original object. The Memento object is a "opaque object" (the object that no one can or should change).

Originator – it is an object that contains the *actual state of an external object is strictly specified type*. Originator is able to create a unique copy of this state and return it wrapped in a Memento. The Originator does not know the history of changes. You can set a concrete state to Originator from the outside, which will be considered as actual. The Originator must make sure that given state corresponds the allowed type of object. Originator may (but not should) have any methods, but they *they can't make changes to the saved object state*.
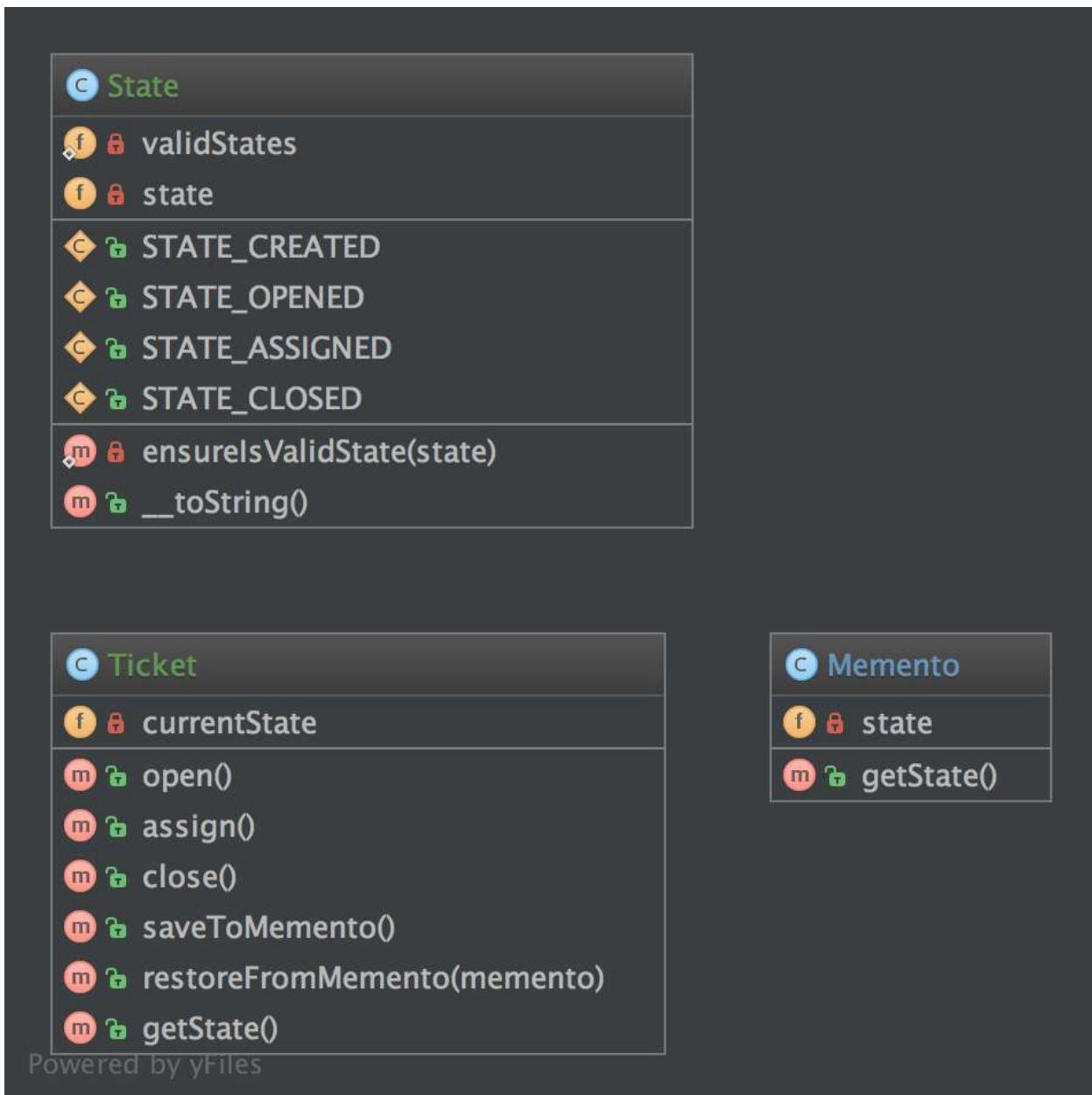
Caretaker *controls the states history*. He may make changes to an object; take a decision to save the state of an external object in the Originator; ask from the Originator snapshot of the current state; or set the Originator state to equivalence with some snapshot from history.

Examples

- The seed of a pseudorandom number generator

- The state in a finite state machine
- Control for intermediate states of ORM Model before saving

UML Diagram

Code

You can also find this code on GitHub

Memento.php

```php
1  <?php
2
3  namespace DesignPatterns\Behavioral\Memento;
4
5  class Memento
6  {
7      /**
8       * @var State
9       */
10     private $state;
11
12     /**
13      * @param State $stateToSave
14      */
15     public function __construct(State $stateToSave)
16     {
17         $this->state = $stateToSave;
18     }
19
20     /**
21      * @return State
22      */
23     public function getState()
24     {
25         return $this->state;
26     }
27 }
```

State.php

```php
1  <?php
2
3  namespace DesignPatterns\Behavioral\Memento;
4
5  class State
6  {
7      const STATE_CREATED = 'created';
8      const STATE_OPENED = 'opened';
9      const STATE_ASSIGNED = 'assigned';
10     const STATE_CLOSED = 'closed';
11
12     /**
13      * @var string
14      */
15     private $state;
16
17     /**
18      * @var string[]
19      */
20     private static $validStates = [
21         self::STATE_CREATED,
22         self::STATE_OPENED,
23         self::STATE_ASSIGNED,
24         self::STATE_CLOSED,
25     ];
26
27     /**
```

(continues on next page)

```php
28         * @param string $state
29         */
30        public function __construct(string $state)
31        {
32            self::ensureIsValidState($state);
33
34            $this->state = $state;
35        }
36
37        private static function ensureIsValidState(string $state)
38        {
39            if (!in_array($state, self::$validStates)) {
40                throw new \InvalidArgumentException('Invalid state given');
41            }
42        }
43
44        public function __toString(): string
45        {
46            return $this->state;
47        }
48    }
```

Ticket.php

```php
1   <?php
2
3   namespace DesignPatterns\Behavioral\Memento;
4
5   /**
6    * Ticket is the "Originator" in this implementation
7    */
8   class Ticket
9   {
10       /**
11        * @var State
12        */
13       private $currentState;
14
15       public function __construct()
16       {
17           $this->currentState = new State(State::STATE_CREATED);
18       }
19
20       public function open()
21       {
22           $this->currentState = new State(State::STATE_OPENED);
23       }
24
25       public function assign()
26       {
27           $this->currentState = new State(State::STATE_ASSIGNED);
28       }
29
30       public function close()
31       {
32           $this->currentState = new State(State::STATE_CLOSED);
33       }
```

```php
34
35     public function saveToMemento(): Memento
36     {
37         return new Memento(clone $this->currentState);
38     }
39
40     public function restoreFromMemento(Memento $memento)
41     {
42         $this->currentState = $memento->getState();
43     }
44
45     public function getState(): State
46     {
47         return $this->currentState;
48     }
49 }
```

Test

Tests/MementoTest.php

```php
1  <?php
2
3  namespace DesignPatterns\Behavioral\Memento\Tests;
4
5  use DesignPatterns\Behavioral\Memento\State;
6  use DesignPatterns\Behavioral\Memento\Ticket;
7  use PHPUnit\Framework\TestCase;
8
9  class MementoTest extends TestCase
10 {
11     public function testOpenTicketAssignAndSetBackToOpen()
12     {
13         $ticket = new Ticket();
14
15         // open the ticket
16         $ticket->open();
17         $openedState = $ticket->getState();
18         $this->assertEquals(State::STATE_OPENED, (string) $ticket->getState());
19
20         $memento = $ticket->saveToMemento();
21
22         // assign the ticket
23         $ticket->assign();
24         $this->assertEquals(State::STATE_ASSIGNED, (string) $ticket->getState());
25
26         // now restore to the opened state, but verify that the state object has been␣
   ↪cloned for the memento
27         $ticket->restoreFromMemento($memento);
28
29         $this->assertEquals(State::STATE_OPENED, (string) $ticket->getState());
30         $this->assertNotSame($openedState, $ticket->getState());
31     }
32 }
```

1.3.6 Null Object

Purpose

NullObject is not a GoF design pattern but a schema which appears frequently enough to be considered a pattern. It has the following benefits:
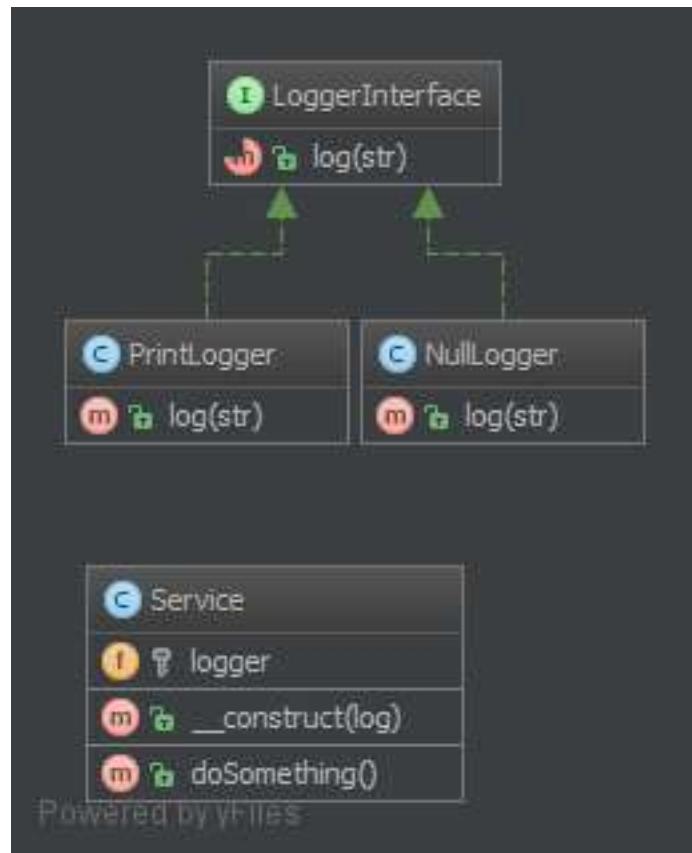
- Client code is simplified
- Reduces the chance of null pointer exceptions
- Fewer conditionals require less test cases

Methods that return an object or null should instead return an object or `NullObject`. `NullObjects` simplify boilerplate code such as `if (!is_null($obj)) { $obj->callSomething(); }` to just `$obj->callSomething();` by eliminating the conditional check in client code.

Examples

- Symfony2: null logger of profiler
- Symfony2: null output in Symfony/Console
- null handler in a Chain of Responsibilities pattern
- null command in a Command pattern

UML Diagram

Code

You can also find this code on GitHub

Service.php

```php
<?php

namespace DesignPatterns\Behavioral\NullObject;

class Service
{
    /**
     * @var LoggerInterface
     */
    private $logger;

    /**
     * @param LoggerInterface $logger
     */
    public function __construct(LoggerInterface $logger)
    {
        $this->logger = $logger;
    }

    /**
     * do something ...
     */
    public function doSomething()
    {
        // notice here that you don't have to check if the logger is set with eg. is_
→null(), instead just use it
        $this->logger->log('We are in '.__METHOD__);
    }
}
```

LoggerInterface.php

```php
<?php

namespace DesignPatterns\Behavioral\NullObject;

/**
 * Key feature: NullLogger must inherit from this interface like any other loggers
 */
interface LoggerInterface
{
    public function log(string $str);
}
```

PrintLogger.php

```php
<?php

namespace DesignPatterns\Behavioral\NullObject;

class PrintLogger implements LoggerInterface
{
```

(continues on next page)

```php
7       public function log(string $str)
8       {
9           echo $str;
10      }
11  }
```

NullLogger.php

```php
1   <?php
2
3   namespace DesignPatterns\Behavioral\NullObject;
4
5   class NullLogger implements LoggerInterface
6   {
7       public function log(string $str)
8       {
9           // do nothing
10      }
11  }
```

Test

Tests/LoggerTest.php

```php
1   <?php
2
3   namespace DesignPatterns\Behavioral\NullObject\Tests;
4
5   use DesignPatterns\Behavioral\NullObject\NullLogger;
6   use DesignPatterns\Behavioral\NullObject\PrintLogger;
7   use DesignPatterns\Behavioral\NullObject\Service;
8   use PHPUnit\Framework\TestCase;
9
10  class LoggerTest extends TestCase
11  {
12      public function testNullObject()
13      {
14          $service = new Service(new NullLogger());
15          $this->expectOutputString('');
16          $service->doSomething();
17      }
18
19      public function testStandardLogger()
20      {
21          $service = new Service(new PrintLogger());
22          $this->expectOutputString('We are in␣
    ↪DesignPatterns\Behavioral\NullObject\Service::doSomething');
23          $service->doSomething();
24      }
25  }
```

1.3.7 Observer

Purpose

To implement a publish/subscribe behaviour to an object, whenever a "Subject" object changes its state, the attached "Observers" will be notified. It is used to shorten the amount of coupled objects and uses loose coupling instead.
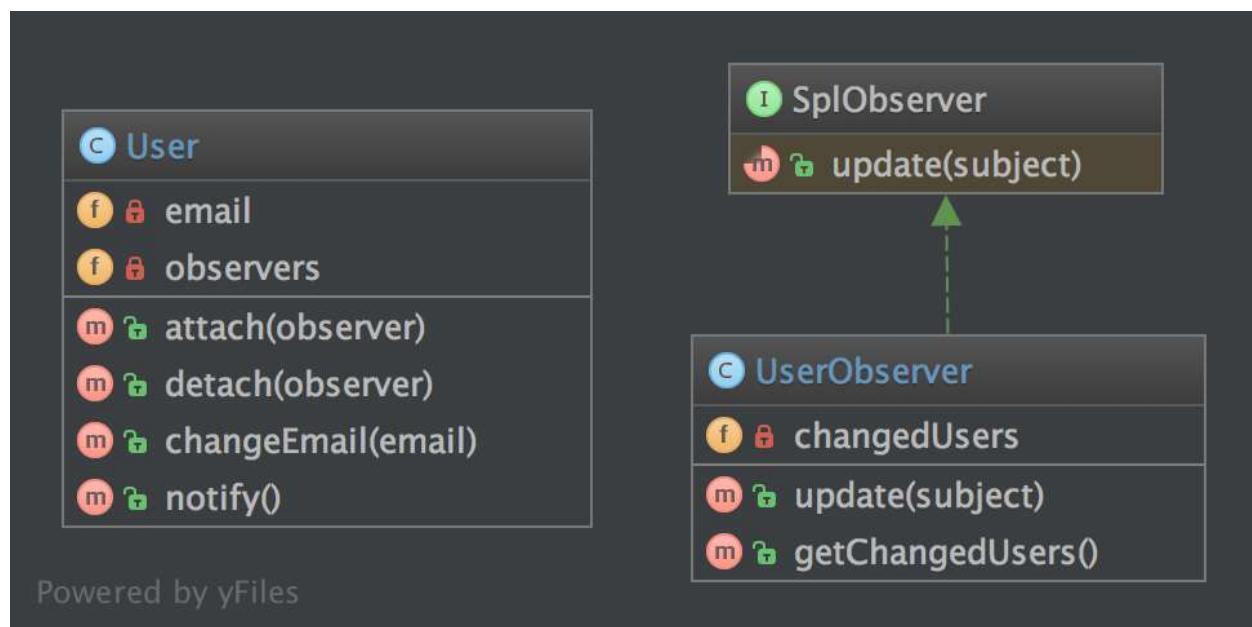
Examples

- a message queue system is observed to show the progress of a job in a GUI

Note

PHP already defines two interfaces that can help to implement this pattern: SplObserver and SplSubject.

UML Diagram

Code

You can also find this code on GitHub

User.php

```php
<?php

namespace DesignPatterns\Behavioral\Observer;

/**
 * User implements the observed object (called Subject), it maintains a list of
 →observers and sends notifications to
 * them in case changes are made on the User object
```

(continues on next page)

```php
 8      */
 9  class User implements \SplSubject
10  {
11      /**
12       * @var string
13       */
14      private $email;
15
16      /**
17       * @var \SplObjectStorage
18       */
19      private $observers;
20
21      public function __construct()
22      {
23          $this->observers = new \SplObjectStorage();
24      }
25
26      public function attach(\SplObserver $observer)
27      {
28          $this->observers->attach($observer);
29      }
30
31      public function detach(\SplObserver $observer)
32      {
33          $this->observers->detach($observer);
34      }
35
36      public function changeEmail(string $email)
37      {
38          $this->email = $email;
39          $this->notify();
40      }
41
42      public function notify()
43      {
44          /** @var \SplObserver $observer */
45          foreach ($this->observers as $observer) {
46              $observer->update($this);
47          }
48      }
49  }
```

UserObserver.php

```php
 1  <?php
 2
 3  namespace DesignPatterns\Behavioral\Observer;
 4
 5  class UserObserver implements \SplObserver
 6  {
 7      /**
 8       * @var User[]
 9       */
10      private $changedUsers = [];
11
12      /**
```

```php
13          * It is called by the Subject, usually by SplSubject::notify()
14          *
15          * @param \SplSubject $subject
16          */
17         public function update(\SplSubject $subject)
18         {
19             $this->changedUsers[] = clone $subject;
20         }
21
22         /**
23          * @return User[]
24          */
25         public function getChangedUsers(): array
26         {
27             return $this->changedUsers;
28         }
29     }
```

Test

Tests/ObserverTest.php

```php
1   <?php
2
3   namespace DesignPatterns\Behavioral\Observer\Tests;
4
5   use DesignPatterns\Behavioral\Observer\User;
6   use DesignPatterns\Behavioral\Observer\UserObserver;
7   use PHPUnit\Framework\TestCase;
8
9   class ObserverTest extends TestCase
10  {
11      public function testChangeInUserLeadsToUserObserverBeingNotified()
12      {
13          $observer = new UserObserver();
14
15          $user = new User();
16          $user->attach($observer);
17
18          $user->changeEmail('foo@bar.com');
19          $this->assertCount(1, $observer->getChangedUsers());
20      }
21  }
```

1.3.8 Specification
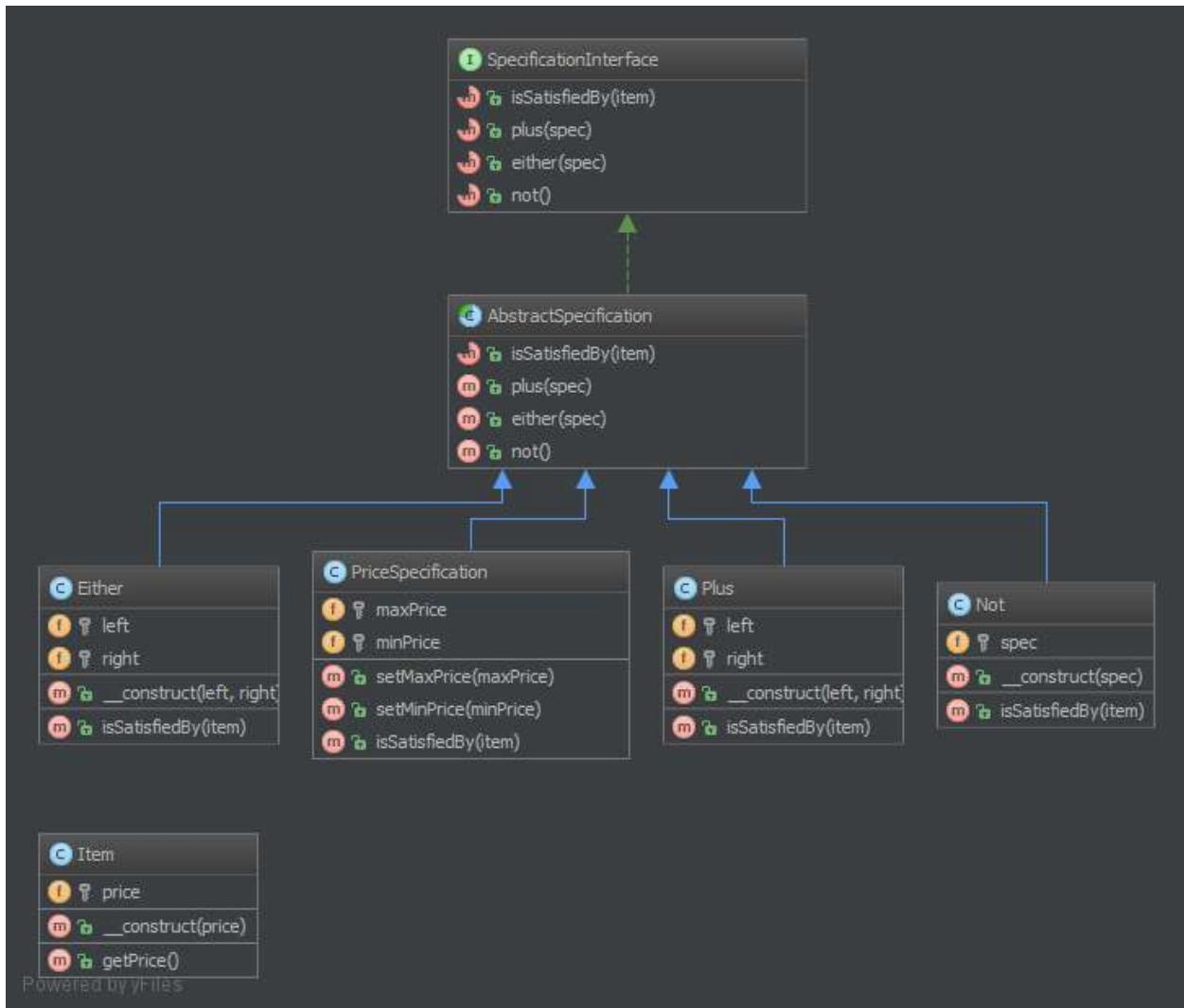
Purpose

Builds a clear specification of business rules, where objects can be checked against. The composite specification class has one method called isSatisfiedBy that returns either true or false depending on whether the given object satisfies the specification.

Examples

- RulerZ

UML Diagram

Code

You can also find this code on GitHub

Item.php

```php
<?php

namespace DesignPatterns\Behavioral\Specification;

class Item
{
```

```php
 7      /**
 8       * @var float
 9       */
10      private $price;
11
12      public function __construct(float $price)
13      {
14          $this->price = $price;
15      }
16
17      public function getPrice(): float
18      {
19          return $this->price;
20      }
21  }
```

SpecificationInterface.php

```php
1  <?php
2
3  namespace DesignPatterns\Behavioral\Specification;
4
5  interface SpecificationInterface
6  {
7      public function isSatisfiedBy(Item $item): bool;
8  }
```

OrSpecification.php

```php
 1  <?php
 2
 3  namespace DesignPatterns\Behavioral\Specification;
 4
 5  class OrSpecification implements SpecificationInterface
 6  {
 7      /**
 8       * @var SpecificationInterface[]
 9       */
10      private $specifications;
11
12      /**
13       * @param SpecificationInterface[] ...$specifications
14       */
15      public function __construct(SpecificationInterface ...$specifications)
16      {
17          $this->specifications = $specifications;
18      }
19
20      /**
21       * if at least one specification is true, return true, else return false
22       */
23      public function isSatisfiedBy(Item $item): bool
24      {
25          foreach ($this->specifications as $specification) {
26              if ($specification->isSatisfiedBy($item)) {
27                  return true;
28              }
```

```php
29          }
30          return false;
31      }
32  }
```

PriceSpecification.php

```php
1   <?php
2
3   namespace DesignPatterns\Behavioral\Specification;
4
5   class PriceSpecification implements SpecificationInterface
6   {
7       /**
8        * @var float|null
9        */
10      private $maxPrice;
11
12      /**
13       * @var float|null
14       */
15      private $minPrice;
16
17      /**
18       * @param float $minPrice
19       * @param float $maxPrice
20       */
21      public function __construct($minPrice, $maxPrice)
22      {
23          $this->minPrice = $minPrice;
24          $this->maxPrice = $maxPrice;
25      }
26
27      public function isSatisfiedBy(Item $item): bool
28      {
29          if ($this->maxPrice !== null && $item->getPrice() > $this->maxPrice) {
30              return false;
31          }
32
33          if ($this->minPrice !== null && $item->getPrice() < $this->minPrice) {
34              return false;
35          }
36
37          return true;
38      }
39  }
```

AndSpecification.php

```php
1   <?php
2
3   namespace DesignPatterns\Behavioral\Specification;
4
5   class AndSpecification implements SpecificationInterface
6   {
7       /**
8        * @var SpecificationInterface[]
```

```php
 9          */
10         private $specifications;
11
12         /**
13          * @param SpecificationInterface[] ...$specifications
14          */
15         public function __construct(SpecificationInterface ...$specifications)
16         {
17             $this->specifications = $specifications;
18         }
19
20         /**
21          * if at least one specification is false, return false, else return true.
22          */
23         public function isSatisfiedBy(Item $item): bool
24         {
25             foreach ($this->specifications as $specification) {
26                 if (!$specification->isSatisfiedBy($item)) {
27                     return false;
28                 }
29             }
30
31             return true;
32         }
33     }
```

NotSpecification.php

```php
 1  <?php
 2
 3  namespace DesignPatterns\Behavioral\Specification;
 4
 5  class NotSpecification implements SpecificationInterface
 6  {
 7      /**
 8       * @var SpecificationInterface
 9       */
10      private $specification;
11
12      public function __construct(SpecificationInterface $specification)
13      {
14          $this->specification = $specification;
15      }
16
17      public function isSatisfiedBy(Item $item): bool
18      {
19          return !$this->specification->isSatisfiedBy($item);
20      }
21  }
```

Test

Tests/SpecificationTest.php

```php
1   <?php
2
3   namespace DesignPatterns\Behavioral\Specification\Tests;
4
5   use DesignPatterns\Behavioral\Specification\Item;
6   use DesignPatterns\Behavioral\Specification\NotSpecification;
7   use DesignPatterns\Behavioral\Specification\OrSpecification;
8   use DesignPatterns\Behavioral\Specification\AndSpecification;
9   use DesignPatterns\Behavioral\Specification\PriceSpecification;
10  use PHPUnit\Framework\TestCase;
11
12  class SpecificationTest extends TestCase
13  {
14      public function testCanOr()
15      {
16          $spec1 = new PriceSpecification(50, 99);
17          $spec2 = new PriceSpecification(101, 200);
18
19          $orSpec = new OrSpecification($spec1, $spec2);
20
21          $this->assertFalse($orSpec->isSatisfiedBy(new Item(100)));
22          $this->assertTrue($orSpec->isSatisfiedBy(new Item(51)));
23          $this->assertTrue($orSpec->isSatisfiedBy(new Item(150)));
24      }
25
26      public function testCanAnd()
27      {
28          $spec1 = new PriceSpecification(50, 100);
29          $spec2 = new PriceSpecification(80, 200);
30
31          $andSpec = new AndSpecification($spec1, $spec2);
32
33          $this->assertFalse($andSpec->isSatisfiedBy(new Item(150)));
34          $this->assertFalse($andSpec->isSatisfiedBy(new Item(1)));
35          $this->assertFalse($andSpec->isSatisfiedBy(new Item(51)));
36          $this->assertTrue($andSpec->isSatisfiedBy(new Item(100)));
37      }
38
39      public function testCanNot()
40      {
41          $spec1 = new PriceSpecification(50, 100);
42          $notSpec = new NotSpecification($spec1);
43
44          $this->assertTrue($notSpec->isSatisfiedBy(new Item(150)));
45          $this->assertFalse($notSpec->isSatisfiedBy(new Item(50)));
46      }
47  }
```
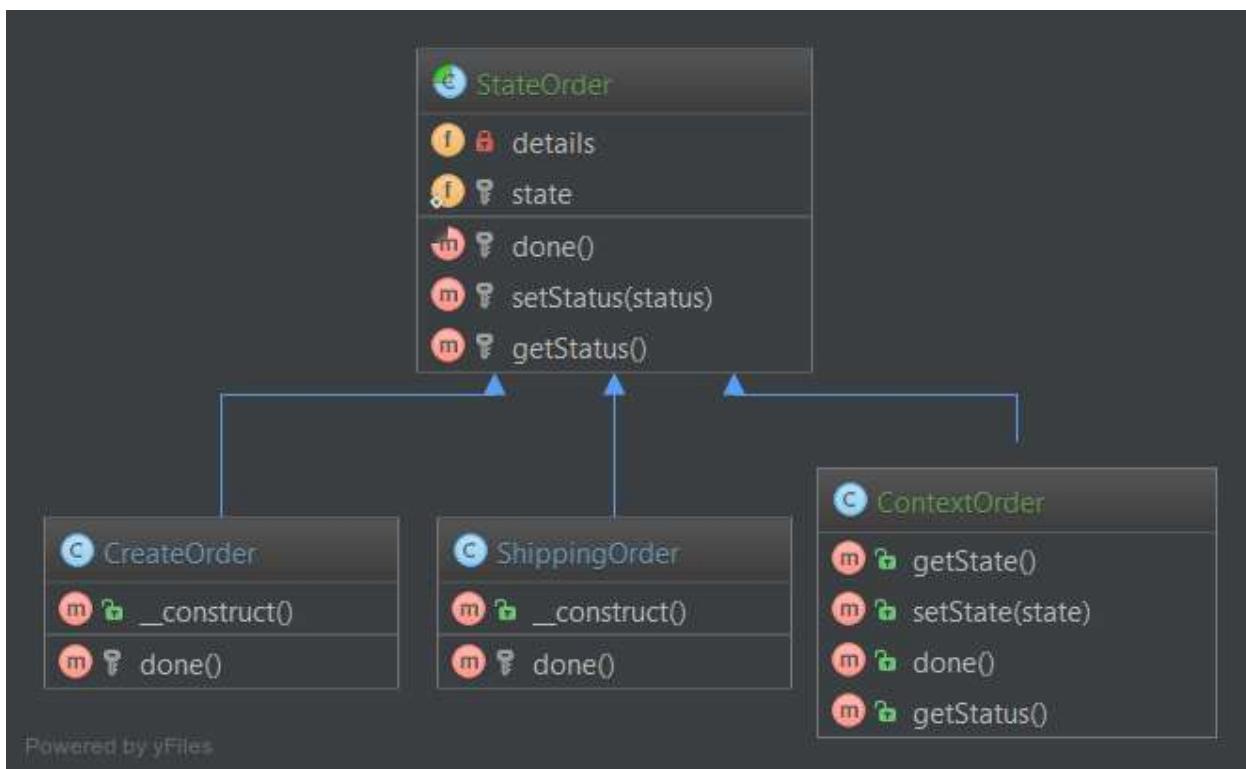
1.3.9 State

Purpose

Encapsulate varying behavior for the same routine based on an object's state. This can be a cleaner way for an object to change its behavior at runtime without resorting to large monolithic conditional statements.

UML Diagram

Code

You can also find this code on GitHub

OrderContext.php

```php
<?php

namespace DesignPatterns\Behavioral\State;

class OrderContext
{
    /**
     * @var State
     */
    private $state;

    public static function create(): OrderContext
    {
        $order = new self();
        $order->state = new StateCreated();

        return $order;
    }

    public function setState(State $state)
    {
        $this->state = $state;
```

```php
23          }
24
25          public function proceedToNext()
26          {
27              $this->state->proceedToNext($this);
28          }
29
30          public function toString()
31          {
32              return $this->state->toString();
33          }
34      }
```

State.php

```php
1   <?php
2
3   namespace DesignPatterns\Behavioral\State;
4
5   interface State
6   {
7       public function proceedToNext(OrderContext $context);
8
9       public function toString(): string;
10  }
```

StateCreated.php

```php
1   <?php
2
3   namespace DesignPatterns\Behavioral\State;
4
5   class StateCreated implements State
6   {
7       public function proceedToNext(OrderContext $context)
8       {
9           $context->setState(new StateShipped());
10      }
11
12      public function toString(): string
13      {
14          return 'created';
15      }
16  }
```

StateShipped.php

```php
1   <?php
2
3   namespace DesignPatterns\Behavioral\State;
4
5   class StateShipped implements State
6   {
7       public function proceedToNext(OrderContext $context)
8       {
9           $context->setState(new StateDone());
10      }
```

```php
11
12      public function toString(): string
13      {
14          return 'shipped';
15      }
16  }
```

StateDone.php

```php
1   <?php
2
3   namespace DesignPatterns\Behavioral\State;
4
5   class StateDone implements State
6   {
7       public function proceedToNext(OrderContext $context)
8       {
9           // there is nothing more to do
10      }
11
12      public function toString(): string
13      {
14          return 'done';
15      }
16  }
```

Test

Tests/StateTest.php

```php
1   <?php
2
3   namespace DesignPatterns\Behavioral\State\Tests;
4
5   use DesignPatterns\Behavioral\State\OrderContext;
6   use PHPUnit\Framework\TestCase;
7
8   class StateTest extends TestCase
9   {
10      public function testIsCreatedWithStateCreated()
11      {
12          $orderContext = OrderContext::create();
13
14          $this->assertEquals('created', $orderContext->toString());
15      }
16
17      public function testCanProceedToStateShipped()
18      {
19          $contextOrder = OrderContext::create();
20          $contextOrder->proceedToNext();
21
22          $this->assertEquals('shipped', $contextOrder->toString());
23      }
24
25      public function testCanProceedToStateDone()
```

```
26      {
27          $contextOrder = OrderContext::create();
28          $contextOrder->proceedToNext();
29          $contextOrder->proceedToNext();
30
31          $this->assertEquals('done', $contextOrder->toString());
32      }
33
34      public function testStateDoneIsTheLastPossibleState()
35      {
36          $contextOrder = OrderContext::create();
37          $contextOrder->proceedToNext();
38          $contextOrder->proceedToNext();
39          $contextOrder->proceedToNext();
40
41          $this->assertEquals('done', $contextOrder->toString());
42      }
43  }
```

1.3.10 Strategy

Terminology:

- Context
- Strategy
- Concrete Strategy
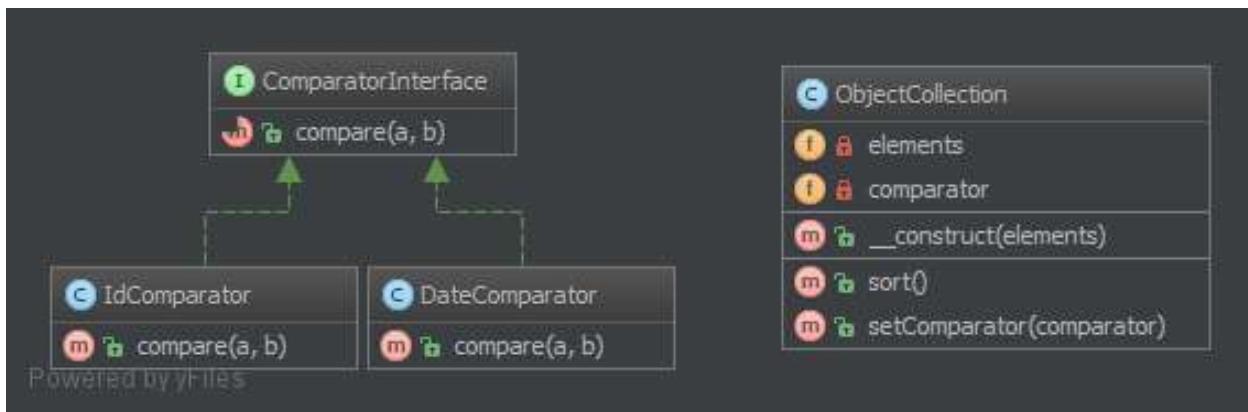
Purpose

To separate strategies and to enable fast switching between them. Also this pattern is a good alternative to inheritance (instead of having an abstract class that is extended).

Examples

- sorting a list of objects, one strategy by date, the other by id
- simplify unit testing: e.g. switching between file and in-memory storage

UML Diagram

Code

You can also find this code on GitHub

Context.php

```php
<?php

namespace DesignPatterns\Behavioral\Strategy;

class Context
{
    /**
     * @var ComparatorInterface
     */
    private $comparator;

    public function __construct(ComparatorInterface $comparator)
    {
        $this->comparator = $comparator;
    }

    public function executeStrategy(array $elements) : array
    {
        uasort($elements, [$this->comparator, 'compare']);

        return $elements;
    }
}
```

ComparatorInterface.php

```php
<?php

namespace DesignPatterns\Behavioral\Strategy;

interface ComparatorInterface
{
    /**
     * @param mixed $a
```

```php
9         * @param mixed $b
10        *
11        * @return int
12        */
13       public function compare($a, $b): int;
14   }
```

DateComparator.php

```php
1    <?php
2
3    namespace DesignPatterns\Behavioral\Strategy;
4
5    class DateComparator implements ComparatorInterface
6    {
7        /**
8         * @param mixed $a
9         * @param mixed $b
10        *
11        * @return int
12        */
13       public function compare($a, $b): int
14       {
15           $aDate = new \DateTime($a['date']);
16           $bDate = new \DateTime($b['date']);
17
18           return $aDate <=> $bDate;
19       }
20   }
```

IdComparator.php

```php
1    <?php
2
3    namespace DesignPatterns\Behavioral\Strategy;
4
5    class IdComparator implements ComparatorInterface
6    {
7        /**
8         * @param mixed $a
9         * @param mixed $b
10        *
11        * @return int
12        */
13       public function compare($a, $b): int
14       {
15           return $a['id'] <=> $b['id'];
16       }
17   }
```

Test

Tests/StrategyTest.php

```php
1   <?php
2
3   namespace DesignPatterns\Behavioral\Strategy\Tests;
4
5   use DesignPatterns\Behavioral\Strategy\Context;
6   use DesignPatterns\Behavioral\Strategy\DateComparator;
7   use DesignPatterns\Behavioral\Strategy\IdComparator;
8   use PHPUnit\Framework\TestCase;
9
10  class StrategyTest extends TestCase
11  {
12      public function provideIntegers()
13      {
14          return [
15              [
16                  [['id' => 2], ['id' => 1], ['id' => 3]],
17                  ['id' => 1],
18              ],
19              [
20                  [['id' => 3], ['id' => 2], ['id' => 1]],
21                  ['id' => 1],
22              ],
23          ];
24      }
25
26      public function provideDates()
27      {
28          return [
29              [
30                  [['date' => '2014-03-03'], ['date' => '2015-03-02'], ['date' => '2013-
    ↪03-01']],
31                  ['date' => '2013-03-01'],
32              ],
33              [
34                  [['date' => '2014-02-03'], ['date' => '2013-02-01'], ['date' => '2015-
    ↪02-02']],
35                  ['date' => '2013-02-01'],
36              ],
37          ];
38      }
39
40      /**
41       * @dataProvider provideIntegers
42       *
43       * @param array $collection
44       * @param array $expected
45       */
46      public function testIdComparator($collection, $expected)
47      {
48          $obj = new Context(new IdComparator());
49          $elements = $obj->executeStrategy($collection);
50
51          $firstElement = array_shift($elements);
52          $this->assertEquals($expected, $firstElement);
53      }
54
55      /**
```

```
56          * @dataProvider provideDates
57          *
58          * @param array $collection
59          * @param array $expected
60          */
61         public function testDateComparator($collection, $expected)
62         {
63             $obj = new Context(new DateComparator());
64             $elements = $obj->executeStrategy($collection);
65
66             $firstElement = array_shift($elements);
67             $this->assertEquals($expected, $firstElement);
68         }
69     }
```

1.3.11 Template Method

Purpose

Template Method is a behavioral design pattern.
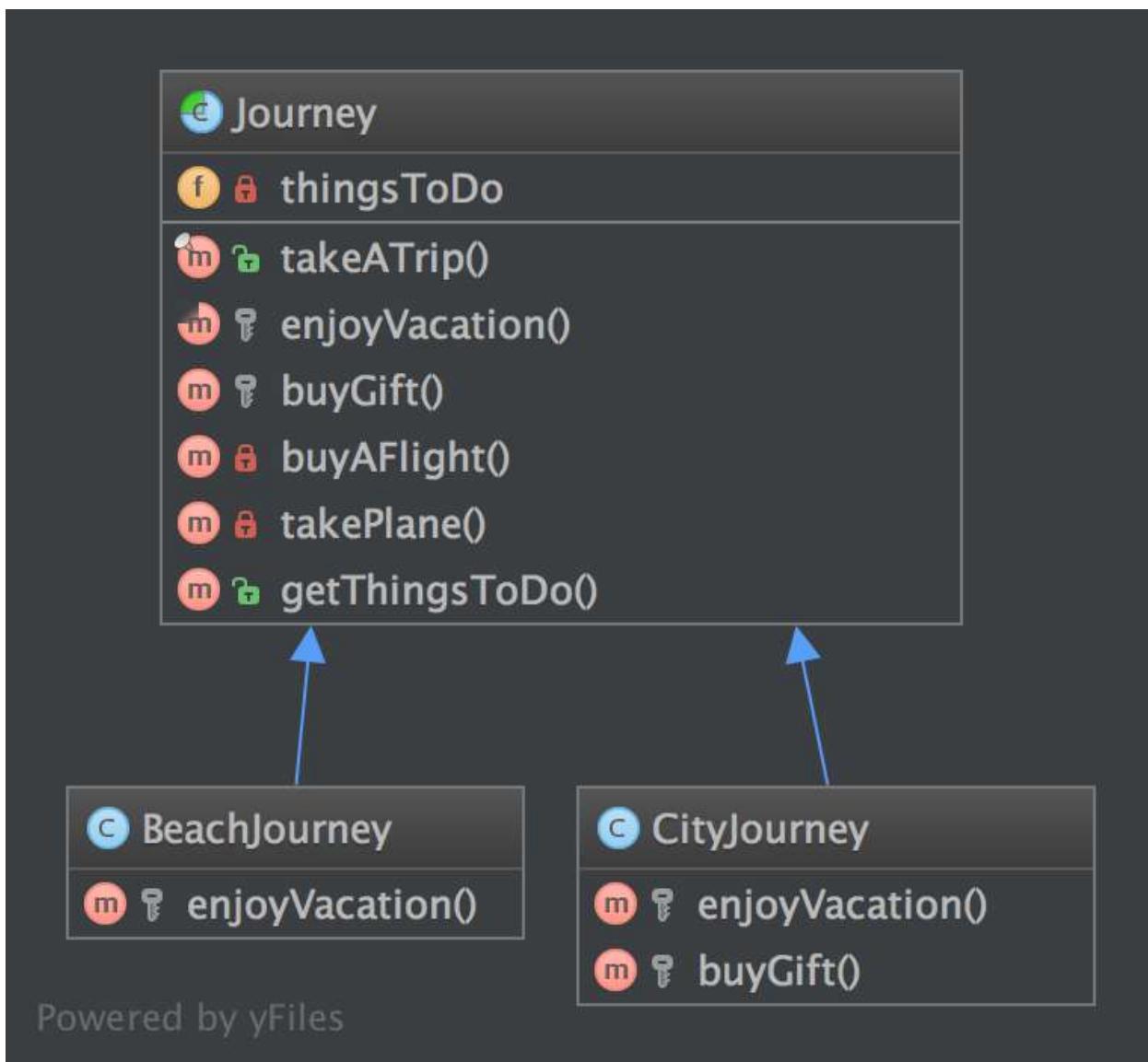
Perhaps you have encountered it many times already. The idea is to let subclasses of this abstract template "finish" the behavior of an algorithm.

A.k.a the "Hollywood principle": "Don't call us, we call you." This class is not called by subclasses but the inverse. How? With abstraction of course.

In other words, this is a skeleton of algorithm, well-suited for framework libraries. The user has just to implement one method and the superclass do the job.

It is an easy way to decouple concrete classes and reduce copy-paste, that's why you'll find it everywhere.

UML Diagram

Code

You can also find this code on GitHub

Journey.php

```php
<?php

namespace DesignPatterns\Behavioral\TemplateMethod;

abstract class Journey
{
    /**
     * @var string[]
```

(continues on next page)

```php
 9          */
10         private $thingsToDo = [];
11
12         /**
13          * This is the public service provided by this class and its subclasses.
14          * Notice it is final to "freeze" the global behavior of algorithm.
15          * If you want to override this contract, make an interface with only takeATrip()
16          * and subclass it.
17          */
18         final public function takeATrip()
19         {
20             $this->thingsToDo[] = $this->buyAFlight();
21             $this->thingsToDo[] = $this->takePlane();
22             $this->thingsToDo[] = $this->enjoyVacation();
23             $buyGift = $this->buyGift();
24
25             if ($buyGift !== null) {
26                 $this->thingsToDo[] = $buyGift;
27             }
28
29             $this->thingsToDo[] = $this->takePlane();
30         }
31
32         /**
33          * This method must be implemented, this is the key-feature of this pattern.
34          */
35         abstract protected function enjoyVacation(): string;
36
37         /**
38          * This method is also part of the algorithm but it is optional.
39          * You can override it only if you need to
40          *
41          * @return null|string
42          */
43         protected function buyGift()
44         {
45             return null;
46         }
47
48         private function buyAFlight(): string
49         {
50             return 'Buy a flight ticket';
51         }
52
53         private function takePlane(): string
54         {
55             return 'Taking the plane';
56         }
57
58         /**
59          * @return string[]
60          */
61         public function getThingsToDo(): array
62         {
63             return $this->thingsToDo;
64         }
65     }
```

BeachJourney.php

```php
<?php

namespace DesignPatterns\Behavioral\TemplateMethod;

class BeachJourney extends Journey
{
    protected function enjoyVacation(): string
    {
        return "Swimming and sun-bathing";
    }
}
```

CityJourney.php

```php
<?php

namespace DesignPatterns\Behavioral\TemplateMethod;

class CityJourney extends Journey
{
    protected function enjoyVacation(): string
    {
        return "Eat, drink, take photos and sleep";
    }

    protected function buyGift(): string
    {
        return "Buy a gift";
    }
}
```

Test

Tests/JourneyTest.php

```php
<?php

namespace DesignPatterns\Behavioral\TemplateMethod\Tests;

use DesignPatterns\Behavioral\TemplateMethod;
use PHPUnit\Framework\TestCase;

class JourneyTest extends TestCase
{
    public function testCanGetOnVacationOnTheBeach()
    {
        $beachJourney = new TemplateMethod\BeachJourney();
        $beachJourney->takeATrip();

        $this->assertEquals(
            ['Buy a flight ticket', 'Taking the plane', 'Swimming and sun-bathing',
→'Taking the plane'],
            $beachJourney->getThingsToDo()
        );
```

```
19        }
20
21        public function testCanGetOnAJourneyToACity()
22        {
23            $beachJourney = new TemplateMethod\CityJourney();
24            $beachJourney->takeATrip();
25
26            $this->assertEquals(
27                [
28                    'Buy a flight ticket',
29                    'Taking the plane',
30                    'Eat, drink, take photos and sleep',
31                    'Buy a gift',
32                    'Taking the plane'
33                ],
34                $beachJourney->getThingsToDo()
35            );
36        }
37    }
```
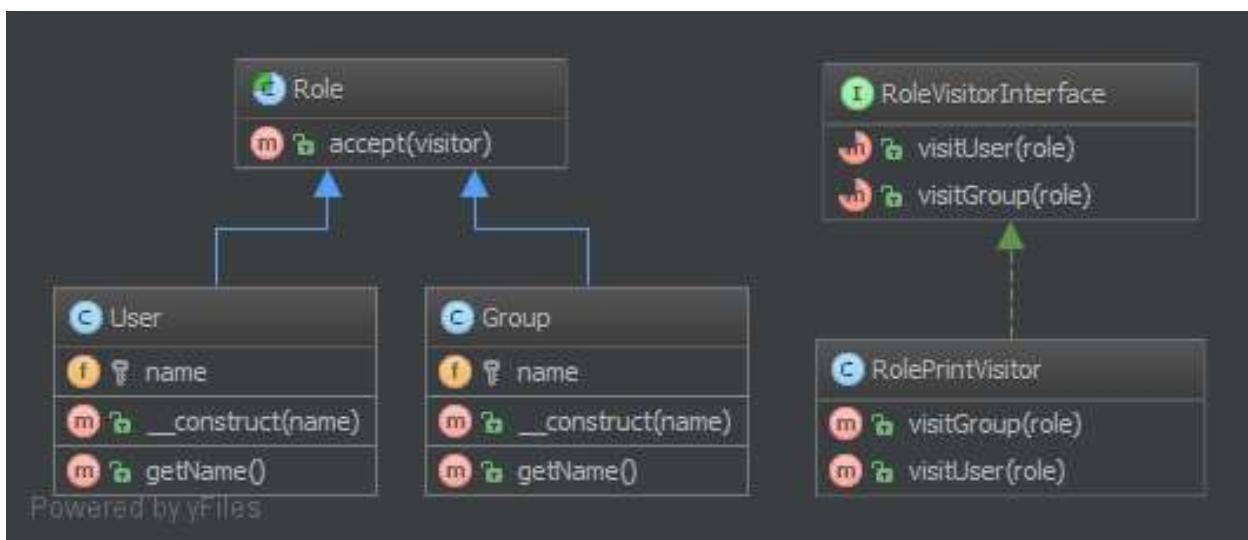
1.3.12 Visitor

Purpose

The Visitor Pattern lets you outsource operations on objects to other objects. The main reason to do this is to keep a separation of concerns. But classes have to define a contract to allow visitors (the `Role::accept` method in the example).

The contract is an abstract class but you can have also a clean interface. In that case, each Visitor has to choose itself which method to invoke on the visitor.

UML Diagram

Code

You can also find this code on GitHub

RoleVisitorInterface.php

```php
<?php

namespace DesignPatterns\Behavioral\Visitor;

/**
 * Note: the visitor must not choose itself which method to
 * invoke, it is the Visitee that make this decision
 */
interface RoleVisitorInterface
{
    public function visitUser(User $role);

    public function visitGroup(Group $role);
}
```

RoleVisitor.php

```php
<?php

namespace DesignPatterns\Behavioral\Visitor;

class RoleVisitor implements RoleVisitorInterface
{
    /**
     * @var Role[]
     */
    private $visited = [];

    public function visitGroup(Group $role)
    {
        $this->visited[] = $role;
    }

    public function visitUser(User $role)
    {
        $this->visited[] = $role;
    }

    /**
     * @return Role[]
     */
    public function getVisited(): array
    {
        return $this->visited;
    }
}
```

Role.php

```php
<?php

namespace DesignPatterns\Behavioral\Visitor;
```

(continues on next page)

```php
4
5   interface Role
6   {
7       public function accept(RoleVisitorInterface $visitor);
8   }
```

User.php

```php
1   <?php
2
3   namespace DesignPatterns\Behavioral\Visitor;
4
5   class User implements Role
6   {
7       /**
8        * @var string
9        */
10      private $name;
11
12      public function __construct(string $name)
13      {
14          $this->name = $name;
15      }
16
17      public function getName(): string
18      {
19          return sprintf('User %s', $this->name);
20      }
21
22      public function accept(RoleVisitorInterface $visitor)
23      {
24          $visitor->visitUser($this);
25      }
26  }
```

Group.php

```php
1   <?php
2
3   namespace DesignPatterns\Behavioral\Visitor;
4
5   class Group implements Role
6   {
7       /**
8        * @var string
9        */
10      private $name;
11
12      public function __construct(string $name)
13      {
14          $this->name = $name;
15      }
16
17      public function getName(): string
18      {
19          return sprintf('Group: %s', $this->name);
20      }
```

```php
21
22      public function accept(RoleVisitorInterface $visitor)
23      {
24          $visitor->visitGroup($this);
25      }
26  }
```

Test

Tests/VisitorTest.php

```php
1   <?php
2
3   namespace DesignPatterns\Tests\Visitor\Tests;
4
5   use DesignPatterns\Behavioral\Visitor;
6   use PHPUnit\Framework\TestCase;
7
8   class VisitorTest extends TestCase
9   {
10      /**
11       * @var Visitor\RoleVisitor
12       */
13      private $visitor;
14
15      protected function setUp()
16      {
17          $this->visitor = new Visitor\RoleVisitor();
18      }
19
20      public function provideRoles()
21      {
22          return [
23              [new Visitor\User('Dominik')],
24              [new Visitor\Group('Administrators')],
25          ];
26      }
27
28      /**
29       * @dataProvider provideRoles
30       *
31       * @param Visitor\Role $role
32       */
33      public function testVisitSomeRole(Visitor\Role $role)
34      {
35          $role->accept($this->visitor);
36          $this->assertSame($role, $this->visitor->getVisited()[0]);
37      }
38  }
```

1.4 More

1.4.1 Service Locator

THIS IS CONSIDERED TO BE AN ANTI-PATTERN!

Service Locator is considered for some people an anti-pattern. It violates the Dependency Inversion principle. Service Locator hides class' dependencies instead of exposing them as you would do using the Dependency Injection. In case of changes of those dependencies you risk to break the functionality of classes which are using them, making your system difficult to maintain.

Purpose

To implement a loosely coupled architecture in order to get better testable, maintainable and extendable code. DI pattern and Service Locator pattern are an implementation of the Inverse of Control pattern.

Usage

With `ServiceLocator` you can register a service for a given interface. By using the interface you can retrieve the service and use it in the classes of the application without knowing its implementation. You can configure and inject the Service Locator object on bootstrap.

Examples

- Zend Framework 2 uses Service Locator to create and share services used in the framework(i.e. EventManager, ModuleManager, all custom user services provided by modules, etc. . .)

UML Diagram

Code

You can also find this code on GitHub

ServiceLocator.php

```php
<?php

namespace DesignPatterns\More\ServiceLocator;

class ServiceLocator
```

```php
6  {
7      /**
8       * @var array
9       */
10     private $services = [];
11
12     /**
13      * @var array
14      */
15     private $instantiated = [];
16
17     /**
18      * @var array
19      */
20     private $shared = [];
21
22     /**
23      * instead of supplying a class here, you could also store a service for an
   ↪interface
24      *
25      * @param string $class
26      * @param object $service
27      * @param bool $share
28      */
29     public function addInstance(string $class, $service, bool $share = true)
30     {
31         $this->services[$class] = $service;
32         $this->instantiated[$class] = $service;
33         $this->shared[$class] = $share;
34     }
35
36     /**
37      * instead of supplying a class here, you could also store a service for an
   ↪interface
38      *
39      * @param string $class
40      * @param array $params
41      * @param bool $share
42      */
43     public function addClass(string $class, array $params, bool $share = true)
44     {
45         $this->services[$class] = $params;
46         $this->shared[$class] = $share;
47     }
48
49     public function has(string $interface): bool
50     {
51         return isset($this->services[$interface]) || isset($this->instantiated[
   ↪$interface]);
52     }
53
54     /**
55      * @param string $class
56      *
57      * @return object
58      */
59     public function get(string $class)
```

```php
60      {
61          if (isset($this->instantiated[$class]) && $this->shared[$class]) {
62              return $this->instantiated[$class];
63          }
64
65          $args = $this->services[$class];
66
67          switch (count($args)) {
68              case 0:
69                  $object = new $class();
70                  break;
71              case 1:
72                  $object = new $class($args[0]);
73                  break;
74              case 2:
75                  $object = new $class($args[0], $args[1]);
76                  break;
77              case 3:
78                  $object = new $class($args[0], $args[1], $args[2]);
79                  break;
80              default:
81                  throw new \OutOfRangeException('Too many arguments given');
82          }
83
84          if ($this->shared[$class]) {
85              $this->instantiated[$class] = $object;
86          }
87
88          return $object;
89      }
90  }
```

LogService.php

```php
1   <?php
2
3   namespace DesignPatterns\More\ServiceLocator;
4
5   class LogService
6   {
7   }
```

Test

Tests/ServiceLocatorTest.php

```php
1   <?php
2
3   namespace DesignPatterns\More\ServiceLocator\Tests;
4
5   use DesignPatterns\More\ServiceLocator\LogService;
6   use DesignPatterns\More\ServiceLocator\ServiceLocator;
7   use PHPUnit\Framework\TestCase;
8
9   class ServiceLocatorTest extends TestCase
```

```php
10  {
11      /**
12       * @var ServiceLocator
13       */
14      private $serviceLocator;
15
16      public function setUp()
17      {
18          $this->serviceLocator = new ServiceLocator();
19      }
20
21      public function testHasServices()
22      {
23          $this->serviceLocator->addInstance(LogService::class, new LogService());
24
25          $this->assertTrue($this->serviceLocator->has(LogService::class));
26          $this->assertFalse($this->serviceLocator->has(self::class));
27      }
28
29      public function testGetWillInstantiateLogServiceIfNoInstanceHasBeenCreatedYet()
30      {
31          $this->serviceLocator->addClass(LogService::class, []);
32          $logger = $this->serviceLocator->get(LogService::class);
33
34          $this->assertInstanceOf(LogService::class, $logger);
35      }
36  }
```

1.4.2 Repository
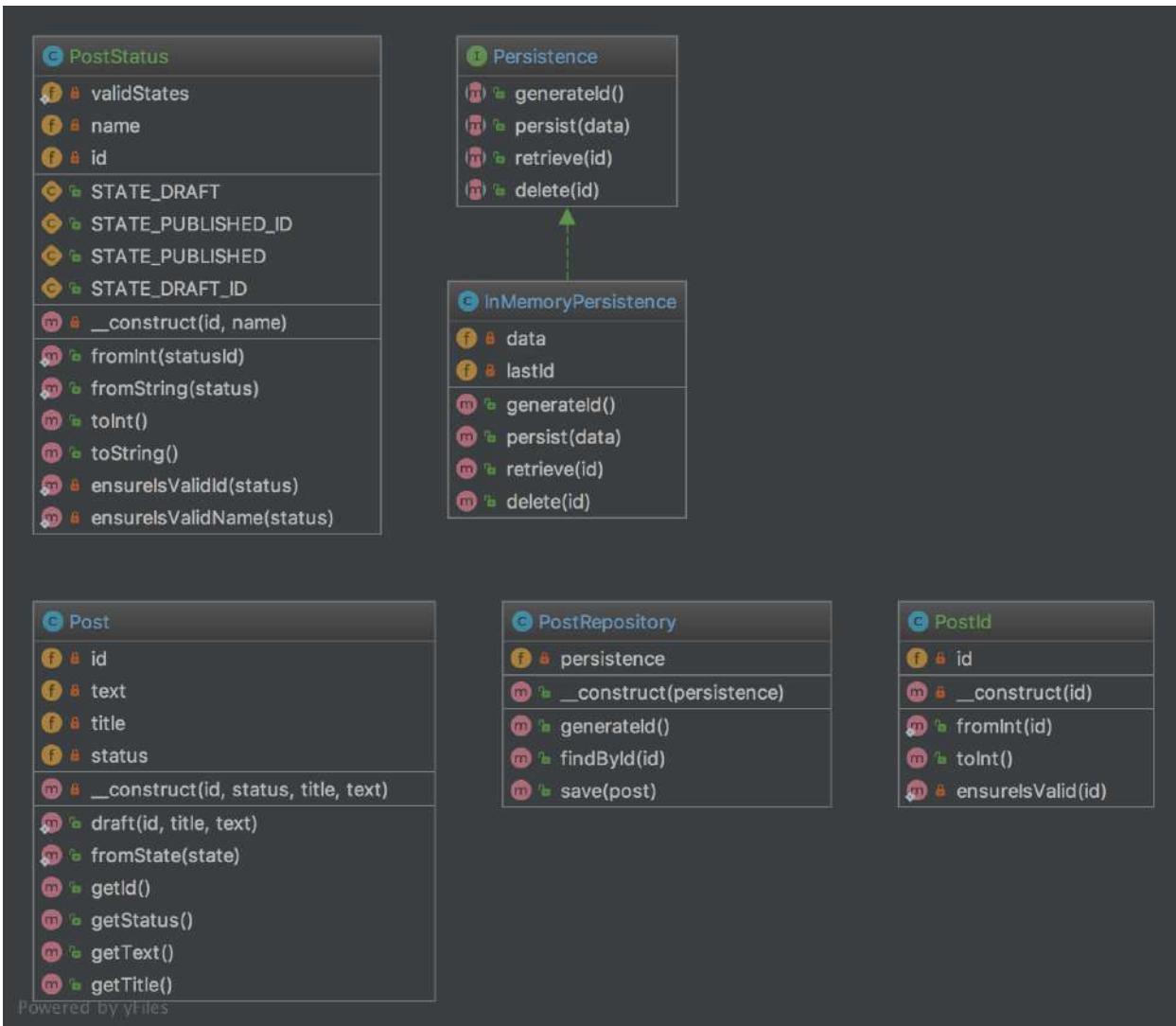
Purpose

Mediates between the domain and data mapping layers using a collection-like interface for accessing domain objects. Repository encapsulates the set of objects persisted in a data store and the operations performed over them, providing a more object-oriented view of the persistence layer. Repository also supports the objective of achieving a clean separation and one-way dependency between the domain and data mapping layers.

Examples

- Doctrine 2 ORM: there is Repository that mediates between Entity and DBAL and contains methods to retrieve objects

- Laravel Framework

UML Diagram

Code

You can also find this code on GitHub

Post.php

```php
<?php

namespace DesignPatterns\More\Repository\Domain;

class Post
{
    /**
     * @var PostId
     */
    private $id;

```

```php
12      /**
13       * @var PostStatus
14       */
15      private $status;
16
17      /**
18       * @var string
19       */
20      private $title;
21
22      /**
23       * @var string
24       */
25      private $text;
26
27      public static function draft(PostId $id, string $title, string $text): Post
28      {
29          return new self(
30              $id,
31              PostStatus::fromString(PostStatus::STATE_DRAFT),
32              $title,
33              $text
34          );
35      }
36
37      public static function fromState(array $state): Post
38      {
39          return new self(
40              PostId::fromInt($state['id']),
41              PostStatus::fromInt($state['statusId']),
42              $state['title'],
43              $state['text']
44          );
45      }
46
47      /**
48       * @param PostId $id
49       * @param PostStatus $status
50       * @param string $title
51       * @param string $text
52       */
53      private function __construct(PostId $id, PostStatus $status, string $title,
    ↪string $text)
54      {
55          $this->id = $id;
56          $this->status = $status;
57          $this->text = $text;
58          $this->title = $title;
59      }
60
61      public function getId(): PostId
62      {
63          return $this->id;
64      }
65
66      public function getStatus(): PostStatus
67      {
```

```php
68          return $this->status;
69      }
70
71      public function getText(): string
72      {
73          return $this->text;
74      }
75
76      public function getTitle(): string
77      {
78          return $this->title;
79      }
80  }
```

PostId.php

```php
1   <?php
2
3   namespace DesignPatterns\More\Repository\Domain;
4
5   /**
6    * This is a perfect example of a value object that is identifiable by it's value
    ↪alone and
7    * is guaranteed to be valid each time an instance is created. Another important
    ↪property of value objects
8    * is immutability.
9    *
10   * Notice also the use of a named constructor (fromInt) which adds a little context
    ↪when creating an instance.
11   */
12  class PostId
13  {
14      /**
15       * @var int
16       */
17      private $id;
18
19      public static function fromInt(int $id)
20      {
21          self::ensureIsValid($id);
22
23          return new self($id);
24      }
25
26      private function __construct(int $id)
27      {
28          $this->id = $id;
29      }
30
31      public function toInt(): int
32      {
33          return $this->id;
34      }
35
36      private static function ensureIsValid(int $id)
37      {
38          if ($id <= 0) {
```

```php
39              throw new \InvalidArgumentException('Invalid PostId given');
40          }
41      }
42  }
```

PostStatus.php

```php
1   <?php
2
3   namespace DesignPatterns\More\Repository\Domain;
4
5   /**
6    * Like PostId, this is a value object which holds the value of the current status of
   ↪a Post. It can be constructed
7    * either from a string or int and is able to validate itself. An instance can then
   ↪be converted back to int or string.
8    */
9   class PostStatus
10  {
11      const STATE_DRAFT_ID = 1;
12      const STATE_PUBLISHED_ID = 2;
13
14      const STATE_DRAFT = 'draft';
15      const STATE_PUBLISHED = 'published';
16
17      private static $validStates = [
18          self::STATE_DRAFT_ID => self::STATE_DRAFT,
19          self::STATE_PUBLISHED_ID => self::STATE_PUBLISHED,
20      ];
21
22      /**
23       * @var int
24       */
25      private $id;
26
27      /**
28       * @var string
29       */
30      private $name;
31
32      public static function fromInt(int $statusId)
33      {
34          self::ensureIsValidId($statusId);
35
36          return new self($statusId, self::$validStates[$statusId]);
37      }
38
39      public static function fromString(string $status)
40      {
41          self::ensureIsValidName($status);
42
43          return new self(array_search($status, self::$validStates), $status);
44      }
45
46      private function __construct(int $id, string $name)
47      {
48          $this->id = $id;
```

```php
49          $this->name = $name;
50      }
51
52      public function toInt(): int
53      {
54          return $this->id;
55      }
56
57      /**
58       * there is a reason that I avoid using __toString() as it operates outside of
     ↪the stack in PHP
59       * and is therefor not able to operate well with exceptions
60       */
61      public function toString(): string
62      {
63          return $this->name;
64      }
65
66      private static function ensureIsValidId(int $status)
67      {
68          if (!in_array($status, array_keys(self::$validStates), true)) {
69              throw new \InvalidArgumentException('Invalid status id given');
70          }
71      }
72
73
74      private static function ensureIsValidName(string $status)
75      {
76          if (!in_array($status, self::$validStates, true)) {
77              throw new \InvalidArgumentException('Invalid status name given');
78          }
79      }
80  }
```

PostRepository.php

```php
1   <?php
2
3   namespace DesignPatterns\More\Repository;
4
5   use DesignPatterns\More\Repository\Domain\Post;
6   use DesignPatterns\More\Repository\Domain\PostId;
7
8   /**
9    * This class is situated between Entity layer (class Post) and access object layer
     ↪(Persistence).
10    *
11    * Repository encapsulates the set of objects persisted in a data store and the
     ↪operations performed over them
12    * providing a more object-oriented view of the persistence layer
13    *
14    * Repository also supports the objective of achieving a clean separation and one-way
     ↪dependency
15    * between the domain and data mapping layers
16    */
17  class PostRepository
18  {
```

```php
19      /**
20       * @var Persistence
21       */
22      private $persistence;
23
24      public function __construct(Persistence $persistence)
25      {
26          $this->persistence = $persistence;
27      }
28
29      public function generateId(): PostId
30      {
31          return PostId::fromInt($this->persistence->generateId());
32      }
33
34      public function findById(PostId $id): Post
35      {
36          try {
37              $arrayData = $this->persistence->retrieve($id->toInt());
38          } catch (\OutOfBoundsException $e) {
39              throw new \OutOfBoundsException(sprintf('Post with id %d does not exist',
    $id->toInt()), 0, $e);
40          }
41
42          return Post::fromState($arrayData);
43      }
44
45      public function save(Post $post)
46      {
47          $this->persistence->persist([
48              'id' => $post->getId()->toInt(),
49              'statusId' => $post->getStatus()->toInt(),
50              'text' => $post->getText(),
51              'title' => $post->getTitle(),
52          ]);
53      }
54  }
```

Persistence.php

```php
1  <?php
2
3  namespace DesignPatterns\More\Repository;
4
5  interface Persistence
6  {
7      public function generateId(): int;
8
9      public function persist(array $data);
10
11     public function retrieve(int $id): array;
12
13     public function delete(int $id);
14 }
```

InMemoryPersistence.php

```php
1  <?php
2
3  namespace DesignPatterns\More\Repository;
4
5  class InMemoryPersistence implements Persistence
6  {
7      /**
8       * @var array
9       */
10     private $data = [];
11
12     /**
13      * @var int
14      */
15     private $lastId = 0;
16
17     public function generateId(): int
18     {
19         $this->lastId++;
20
21         return $this->lastId;
22     }
23
24     public function persist(array $data)
25     {
26         $this->data[$this->lastId] = $data;
27     }
28
29     public function retrieve(int $id): array
30     {
31         if (!isset($this->data[$id])) {
32             throw new \OutOfBoundsException(sprintf('No data found for ID %d', $id));
33         }
34
35         return $this->data[$id];
36     }
37
38     public function delete(int $id)
39     {
40         if (!isset($this->data[$id])) {
41             throw new \OutOfBoundsException(sprintf('No data found for ID %d', $id));
42         }
43
44         unset($this->data[$id]);
45     }
46 }
```

Test

Tests/PostRepositoryTest.php

```php
1  <?php
2
3  namespace DesignPatterns\More\Repository\Tests;
4
5  use DesignPatterns\More\Repository\Domain\PostId;
```

(continues on next page)

```php
6   use DesignPatterns\More\Repository\Domain\PostStatus;
7   use DesignPatterns\More\Repository\InMemoryPersistence;
8   use DesignPatterns\More\Repository\Domain\Post;
9   use DesignPatterns\More\Repository\PostRepository;
10  use PHPUnit\Framework\TestCase;
11
12  class PostRepositoryTest extends TestCase
13  {
14      /**
15       * @var PostRepository
16       */
17      private $repository;
18
19      protected function setUp()
20      {
21          $this->repository = new PostRepository(new InMemoryPersistence());
22      }
23
24      public function testCanGenerateId()
25      {
26          $this->assertEquals(1, $this->repository->generateId()->toInt());
27      }
28
29      /**
30       * @expectedException \OutOfBoundsException
31       * @expectedExceptionMessage Post with id 42 does not exist
32       */
33      public function testThrowsExceptionWhenTryingToFindPostWhichDoesNotExist()
34      {
35          $this->repository->findById(PostId::fromInt(42));
36      }
37
38      public function testCanPersistPostDraft()
39      {
40          $postId = $this->repository->generateId();
41          $post = Post::draft($postId, 'Repository Pattern', 'Design Patterns PHP');
42          $this->repository->save($post);
43
44          $this->repository->findById($postId);
45
46          $this->assertEquals($postId, $this->repository->findById($postId)->getId());
47          $this->assertEquals(PostStatus::STATE_DRAFT, $post->getStatus()->toString());
48      }
49  }
```
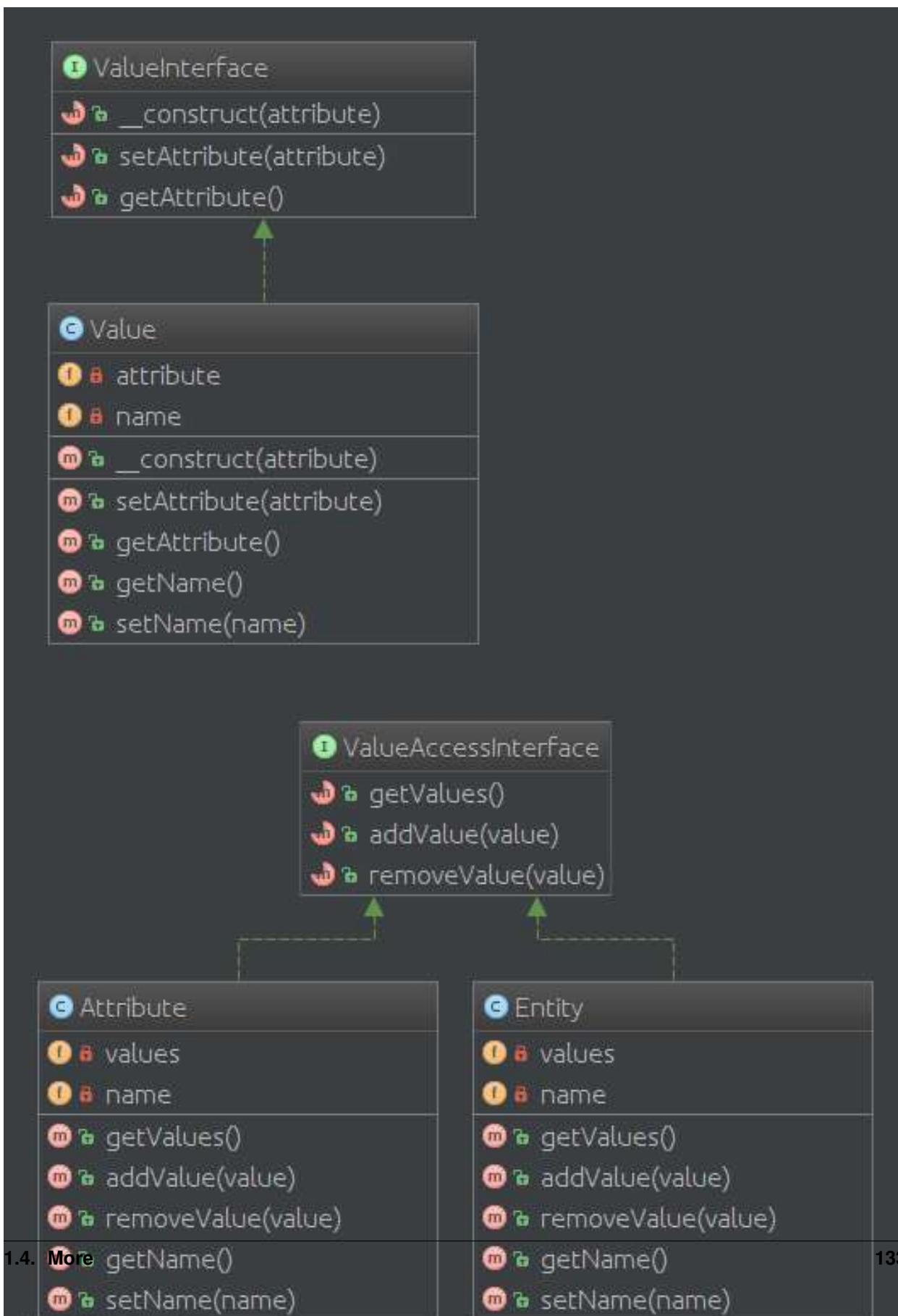
1.4.3 Entity-Attribute-Value (EAV)

The Entity–attribute–value (EAV) pattern in order to implement EAV model with PHP.

Purpose

The Entity–attribute–value (EAV) model is a data model to describe entities where the number of attributes (properties, parameters) that can be used to describe them is potentially vast, but the number that will actually apply to a given entity is relatively modest.

UML Diagram

Code

You can also find this code on GitHub

Entity.php

```php
<?php

namespace DesignPatterns\More\EAV;

class Entity
{
    /**
     * @var \SplObjectStorage
     */
    private $values;

    /**
     * @var string
     */
    private $name;

    /**
     * @param string $name
     * @param Value[] $values
     */
    public function __construct(string $name, $values)
    {
        $this->values = new \SplObjectStorage();
        $this->name = $name;

        foreach ($values as $value) {
            $this->values->attach($value);
        }
    }

    public function __toString(): string
    {
        $text = [$this->name];

        foreach ($this->values as $value) {
            $text[] = (string) $value;
        }

        return join(', ', $text);
    }
}
```

Attribute.php

```php
<?php

namespace DesignPatterns\More\EAV;

class Attribute
{
    /**
     * @var \SplObjectStorage
```

```php
 9          */
10         private $values;
11
12         /**
13          * @var string
14          */
15         private $name;
16
17         public function __construct(string $name)
18         {
19             $this->values = new \SplObjectStorage();
20             $this->name = $name;
21         }
22
23         public function addValue(Value $value)
24         {
25             $this->values->attach($value);
26         }
27
28         /**
29          * @return \SplObjectStorage
30          */
31         public function getValues(): \SplObjectStorage
32         {
33             return $this->values;
34         }
35
36         public function __toString(): string
37         {
38             return $this->name;
39         }
40     }
```

Value.php

```php
 1  <?php
 2
 3  namespace DesignPatterns\More\EAV;
 4
 5  class Value
 6  {
 7      /**
 8       * @var Attribute
 9       */
10      private $attribute;
11
12      /**
13       * @var string
14       */
15      private $name;
16
17      public function __construct(Attribute $attribute, string $name)
18      {
19          $this->name = $name;
20          $this->attribute = $attribute;
21
22          $attribute->addValue($this);
```

```php
23      }
24
25      public function __toString(): string
26      {
27          return sprintf('%s: %s', $this->attribute, $this->name);
28      }
29  }
```

Test

Tests/EAVTest.php

```php
1   <?php
2
3   namespace DesignPatterns\More\EAV\Tests;
4
5   use DesignPatterns\More\EAV\Attribute;
6   use DesignPatterns\More\EAV\Entity;
7   use DesignPatterns\More\EAV\Value;
8   use PHPUnit\Framework\TestCase;
9
10  class EAVTest extends TestCase
11  {
12      public function testCanAddAttributeToEntity()
13      {
14          $colorAttribute = new Attribute('color');
15          $colorSilver = new Value($colorAttribute, 'silver');
16          $colorBlack = new Value($colorAttribute, 'black');
17
18          $memoryAttribute = new Attribute('memory');
19          $memory8Gb = new Value($memoryAttribute, '8GB');
20
21          $entity = new Entity('MacBook Pro', [$colorSilver, $colorBlack, $memory8Gb]);
22
23          $this->assertEquals('MacBook Pro, color: silver, color: black, memory: 8GB',
    →(string) $entity);
24      }
25  }
```

Contribute

If you encounter any bugs or missing translations, please feel free to fork and send a pull request with your changes.
To establish a consistent code quality, please check your code using PHP CodeSniffer against PSR2 standard using
`./vendor/bin/phpcs -p --standard=PSR2 --ignore=vendor ..`